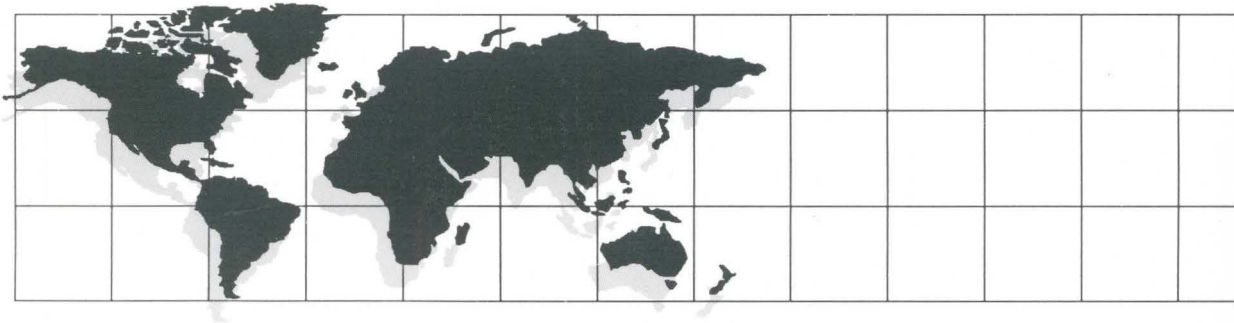


CTOS

Microsoft C

Programming and Installation Reference Manual Volume 1

UNISYS

UNISYS

CTOS®

Microsoft C

Programming and Installation Reference Manual Volume 1

Copyright © 1992 Unisys Corporation
All rights reserved.
Unisys is a registered trademark of Unisys Corporation.
CTOS is a registered trademark of Convergent Technologies, Inc., a wholly owned subsidiary of Unisys Corporation.

Release 6.1

Priced Item

September 1992
Printed in US America
4393 1609-000

The names, places and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any indivdual, living or otherwise, or that of any group or association is purely coincidental and unintended.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a User Communication Form (UCF) with the CLASS specified as "Documentation", the Type specified as "Trouble Report", and the product specified as the title and part number of the manual (for example, 4393 1609-000).

Shared Resource Processor and SRP are trademarks, and CTOS and SuperGen are a registered trademarks of Convergent Technologies Inc., a wholly owned subsidiary of Unisys Corporation.

Z8000 is a registered trademark of Zilog, Inc.

MAPPER, OFIS, and Unisys are registered trademarks, and CUSTOMCARE and MAPPER are registered service marks of Unisys Corporation. BTOS is a trademark of Unisys Corporation.

PDP-11 and VAX-11 are registered trademarks of Digital Equipment Corporation.

Intel is a registered trademark of Intel Corporation.

Page Status

Page	Issue
iii through iv	Original
v through xxxv	Original
xxxvi	Blank
1-1 through 1-12	Original
2-1 through 2-16	Original
3-1 through 3-69	Original
3-70	Blank
4-1 through 4-25	Original
4-26	Blank
5-1 through 5-44	Original
6-1 through 6-33	Original
6-34	Blank
7-1 through 7-38	Original
8-1 through 8-21	Original
8-22	Blank
9-1 through 9-15	Original
9-16	Blank
10-1 through 10-34	Original
11-1 through 11-34	Original
12-1 through 12-37	Original
12-38	Blank
13-1 through 13-33	Original
13-34	Blank
14-1 through 14-43	Original
14-44	Blank
15-1 through 15-32	Original
16-1 through 16-27	Original
16-28	Blank
A-1 through A-31	Original
A-32	Blank
B-1 through B-45	Original
B-46	Blank
C-1 through C-30	Original
D-1 through D-8	Original
E-1 through E-6	Original

Page Status

Page	Issue
F-1	Original
F-2	Blank
G-1 through G-49	Original
G-50	Blank
Glossary-1 through 24	Original
Index-1 through 39	Original
Index-40	Blank

About This Manual

CTOS Microsoft C allows you to write, compile, link, debug, and run C programs. This reference manual contains information about installing the software, writing C programs, and using the various utilities supplied with the CTOS Microsoft C package.

Who Should Use This Manual

This manual is for experienced C programmers who are familiar with CTOS.

How to Use This Manual

If you are using CTOS Microsoft C for the first time, you should read Sections 1 and 2. Section 1 contains overviews of the utilities. Section 2 contains installation instructions. If you are migrating to CTOS Microsoft C from BTOS C, read Appendix A.

If you scan the table of contents and review the topics before you start, you will find this manual easier to use. To find definitions of unfamiliar terms, use the glossary; to locate specific information, use the index.

How This Manual Is Arranged

The material in this manual is divided into 16 sections and seven appendixes. It also contains a glossary and an index.

The material is in two binders. Both binders have a complete Table of Contents, Glossary, and Index that covers material in both binders.

- Section 1 -- Overview

  Provides an overview of the utilities and how to create run files.

- Section 2 -- Installation and Configuration

  Contains installation procedures, and describes how to configure your system for Task Management Service and Environment Service.

- Section 3 -- Using the Programmer's WorkBench

  Describes the basics of using the Programmer's Workbench to edit source code and build applications. Includes procedures for using the Browser and CodeView debuggers.

- Section 4 -- Using Online Help

  Describes how to display and print Help screens using QuickHelp and the Advisor.

- Section 5 -- Building Applications

  Contains details on how to build CTOS applications with the compiler and linker. Describes required startup object modules and libraries to use with the linker.

- Section 6 -- Optimizing C Programs

  Explains compiler and linker optimizations you can use when building applications.

- Section 7 -- Managing Memory

  Explains the different memory models, one of which you must choose when compiling, and explains each model's pointer characteristics.

- Section 8 -- Using the In-Line Assembler

  Describes the advantages of the in-line assembler, and how to handle blocks of assembly code in C programs.

- Section 9 -- Controlling Floating-Point Math Operations

  Describes the different floating-point compiler options and how they affect floating-point computations.

- Section 10 -- Managing Development Projects with NMAKE

  Describes how create description files and use NMAKE syntax.

- Section 11 -- Creating Help Files with HelpMake

  Explains how to create, update, and decode Help files and provides a list of Help Application Programming Interface calls.

- Section 12 -- Customizing the Programmer's WorkBench

  Describes how to assign keystrokes to functions, write macros, and create PWB extensions.

- Section 13 -- Debugging C Programs with CodeView

  Explains how to approach debugging programs with the CodeView debugger.

- Section 14 -- Programming with Mixed Languages

  Explains how to write functions in other languages, and incorporate them into your C programs.

- Section 15 -- Writing Portable Programs

  Reveals portability issues, and hardware and compiler assumptions to recognize when writing programs across different platforms.

- Section 16 -- Building CTOS III Applications

  Explains how to create multithread applications and dynamic-link libraries.

- Appendix A -- Migrating to CTOS Microsoft C

  Describes differences to be aware of between BTOS C and CTOS Microsoft C when migrating between the languages.

- Appendix B -- Command Reference

  Shows the command-line options available for all utilities.

- Appendix C -- Implementation-Defined Behavior

  Lists C language implementation areas left open by the American National Standards Institute (ANSI), and describes how CTOS Microsoft C handles these open areas.

- Appendix D -- Compiler Limits and Numerical Ranges

  Lists details about the compiler.

- Appendix E -- printf/scanf Format Specifiers

  Lists format syntax for printf and scanf along with permitted values.

- Appendix F -- Re-entrant Run-Time Library Functions

  Lists re-entrant C run-time library functions.

- Appendix G -- Sample NMAKE Description Files

  Contains examples of NMAKE description files

Conventions

The following conventions apply throughout this manual:

- Keys to press appear in uppercase boldface type (for example, F1 and RETURN).
- When two keys are pressed together, a hyphen separates the keys (for example, ACTION-GO).
- Words in italics act as placeholders for information you supply (for example, *your_source_file*.c).
- Names of commands to enter on the Executive command line appear in boldface type (for example, **MCBIND**).
- Commands you select from pull-down bars are separated from subsequent selections with commas, and both appear in boldface type (for example, **File, Open** means to select **File** from the menu bar, and then select **Open**).
- Optional material appears in brackets (for example,[*option*]).
- Braces and a vertical bar indicate a choice between two or more options (for example, #pragma pack {1 | 2}).

  You must select one of these options unless brackets surround the braces.

- Ellipsis indicate that more items of the same form may follow (for example, [*files*...]).

- Vertical ellipsis indicates that part of the sample program is intentionally omitted. For example:

```
while()
{
  .
  .
  .
}
```

- The term "character" includes the space.

Example Programs

Short example code sequences and occasional complete example programs appear throughout this manual.

Related Product Information

For information about how to use the CTOS Linker, see the *CTOS Development Utilities Reference Manual.*

For information about the CTOS operating system, see the *CTOS Operating System Concepts Manual.*

For information about general CTOS programming techniques and the extended system services provided with the CTOS operating system, see the *CTOS Programming Guide* and *CTOS/Open Programming Practices.*

For information on programming operations for real mode and protected mode versions of CTOS II 3.3 and CTOS/XE 3.1, see the *CTOS Procedural Interface Reference Manual.*

For an explanation of CTOS Executive commands and using the Executive command line, see the *CTOS Executive Reference Manual* and the *CTOS Executive Operations Guide*, respectively.

For information on running multiple application contexts under CTOS Context/Window Manager, see the *CTOS Context / Window Manager Installation and Configuration Guide, Volume 2: Protected Mode.*

For release-specific technical details and ordering information, see the *CTOS Microsoft C 6.1.0 Software Release Announcement.*

For information on BTOS C, see the *BTOS C Compiler Programming Reference Manual.*

For information about Environment Service, refer to the *Environment Service Administrations and Programming Guide*, overpacked with this product.

Note: *Much of the information contained in the Microsoft C Reference in previous versions of Microsoft C now appears in the QuickHelp online reference databases instead.*

Contents

About This Manual v

Section 1. Overview

Compiler and Linker 1-2
NMAKE 1-3
The Programmer's WorkBench 1-5
The CodeView Debugger 1-6
The Browser 1-7
Task Management Service 1-7
Online Help Utilities 1-9
Librarian 1-10
Environment Service 1-11
New Features 1-11

Section 2. Installation and Configuration

System Requirements 2-1
Before You Install 2-2
Preliminary Operations 2-2
Installation Decisions 2-2
Public 2-3
Verbose 2-4
Backup Previous Version 2-4
Save Copy of Backup 2-4
Save Defaults in User File 2-5
Software Destination [sys] 2-5
User Name, CM Configfile Name, Command File Name 2-5
Selecting Software and Directories 2-5
Additional Software 2-8
Installing CTOS Microsoft C 2-8
Installing from Diskettes 2-9
Installing from Tape 2-9
Installing from Your Server Workstation 2-10

Installing Environment Service and Task Management Service 2-10
Configuring Your System 2-11
Installing Task Management Service 2-11
Setting up Environment Service 2-12
Sample EnvironmentConfig.Sys 2-13
The CodeView Debugger 2-14
CTOS Libraries 2-14
Video Note 2-14
Increasing the Maximum Number of Open Files 2-15
Increasing File Handles 2-15
Increasing Streams 2-16
Using the Modified Files 2-16

Section 3. Using the Programmer's WorkBench

Starting PWB 3-2
Using Windows and Menus 3-4
Windows 3-4
Menus 3-8
Overview of Menu Commands 3-8
Choosing Menu Commands 3-14
Closing Menus 3-14
Using Shortcut Keys 3-15
Shaded Commands 3-15
Dialog Boxes 3-15
Getting Help 3-18
Menu Help 3-18
Dialog Box Help 3-18
The Help Menu 3-19
Using the Editor 3-19
Moving around in a Source File 3-20
Defining a Block 3-21
Setting Bookmarks 3-22
Setting Anchors 3-23
Searching for and Changing Text 3-24
Creating Macros 3-26
Customizing the Editor 3-27
Changing Editor Settings 3-28
Modifying Keyboard Assignments 3-29
Using Advanced Editor Features 3-30

Compiling, Linking, and Running Programs 3-30
Procedural Overview 3-31
Sample Program 3-34
Saving Source Files 3-35
Setting Compiler Options 3-35
Setting Linker Options 3-39
Saving Compiler and Linker Options as a Build Option .. 3-41
Creating and Selecting Makefiles 3-43
Notes on Selecting Makefiles 3-45
Selecting a Build Option 3-46
Building Programs 3-47
Running Programs 3-48
Building Programs from Multiple Source Files 3-48
Two Sample Source Files 3-49
Rebuilding IO.Run with Two Source Files 3-50
Quick Reference 3-52
Debugging Programs 3-53
Working with Error Messages 3-54
Using the Browser 3-55
Preparing a Browser Build 3-55
Inspecting with the Browser 3-57
Using the CodeView Debugger 3-61
Preparing a Debug Build 3-62
Starting and Running CodeView 3-64
Debugging IO.c 3-69

Section 4. Using Online Help

Running QuickHelp 4-2
Starting QuickHelp 4-2
Exiting QuickHelp 4-4
Structure of the Help System 4-5
Using Pull-Down Menus and Dialog Boxes 4-7
Pull-Down Menus 4-7
Dialog Boxes 4-15
Navigating through Help Screens 4-15
Navigating with QuickHelp Pull-Down Menus 4-16
Navigating with Hyperlink Commands 4-17
Displaying Hyperlink Topics 4-17
Copying and Pasting 4-18
Programmer's WorkBench 4-19
QuickHelp 4-19

Displaying Keyword Information 4-21
Displaying Error Code Information 4-23
Printing QuickHelp Information 4-24
Configuring a Context Manager Partition 4-25

Section 5. Building Applications

Using the Compiler 5-2
CL Command Form 5-2
Notes on Command Line Options 5-3
CL's MASM Successor Call 5-3
Compiling for Overlays 5-4
Overlay Incompatibilities 5-5
Syntax Errors 5-5
Using the Linker 5-6
Using MLINK 5-7
Command Overview 5-7
Controlling Automatic Linking 5-9
Using an MLINK Response File 5-12
MLINK Defaults 5-12
Using MCBIND 5-14
Cross References between MLINK and MCBIND Options 5-15
Creating a Typical Run File 5-19
Startup and Flavor Object Modules 5-21
Specifying Standard Libraries 5-22
Linking a Minimum-Size Run File 5-24
Linking Floating-point Support 5-25
Creating and Naming Combined Libraries 5-25
Startup Object Modules and Heap Architecture 5-27
Allocating Near Heap Memory with malloc( ) and _nmalloc( ) 5-27
Sizing Near Heap Memory 5-27
Placing Near Heap Memory in DGroup: Three Options .. 5-28
Stack Sharing 5-29
Primary Area with Stack Sharing Overflow 5-30
Primary Area Only 5-30
Sizing and Allocating Memory in Far Heap 5-30
Allocating Far Heap Memory with malloc( ), _fmalloc( ), and halloc( ) 5-31
LDT-based Segments 5-31
Reserving More Segments with MCMEMORY.Obj . 5-32
Linking for Real Mode with _huge Memory 5-32

Using the Environment Service 5-33
Notes on Makefiles 5-34
Two Types of Makefiles 5-34
NMAKE with Batch 5-35
Programming Notes 5-36
Using the _plm Keyword 5-36
_plm Keyword Limitations 5-37
Output Redirection 5-39
Input Redirection 5-39
Handling Nameless Structs and Unions 5-40
Manipulating Error Messages 5-40
Binary and Text Modes 5-41
Miscellaneous Function Behaviors 5-42
Functions Not Supported 5-42

Section 6. Optimizing C Programs

How to Specify Compiler Optimization 6-1
Optimizing from Programmer's WorkBench 6-2
Optimizing from the Command Line 6-3
Optimizing from Source Code with Pragmas 6-3
Default Optimizations 6-5
Common Subexpression Elimination 6-5
Dead-Store Elimination 6-6
Constant Propagation 6-6
Customizing Optimizations 6-7
Choosing Speed or Size (/Ot and /Os) 6-8
Generating Intrinsic Functions (/Oi) 6-8
Assuming No Aliasing (/Oa and /Ow) 6-12
Performing Loop Optimizations (/Ol) 6-15
Disabling Unsafe Loop Optimizations (/On) 6-16
Enabling Aggressive Optimizations (/Oz) 6-16
Removing Stack Probes (/Gs) 6-18
Enabling Global Register Allocation (/Oe) 6-18
Enabling Common Subexpression Optimization (/Oc and /Og) 6-20
Achieving Consistent Floating-Point Results (/Op) 6-20
Using the 80186, 80188, or 80286 Processor (/G0, /G1, /G2) 6-21
Optimizing for Maximum Efficiency (/Ox) 6-22
Function Prolog and Epilog Optimization (/Ou and /Oy) .. 6-23

CTOS Linker Optimizations 6-24
Linker Configuration File 6-24
Enabling Far Call Optimization (FarCallTranslation) 6-25
How FarCallTranslation Affects Your Code 6-26
Benefits of FarCallTranslation 6-26
Packing Code (PackCode) 6-28
Choosing Function-Calling Conventions 6-29
The C Calling Convention (/Gd) 6-29
The PLM/Pascal Calling Convention (/Gc) 6-30
The Register Calling Convention (/Gr) 6-30
Argument-Passing Convention 6-31
Return Value Convention 6-32
Stack Adjustment Convention 6-32
Register Preservation Requirement 6-32
Function-Naming Convention 6-33

Section 7. Managing Memory

Pointers and 64K Segments 7-1
Near Pointers 7-2
Far Pointers 7-3
Huge Pointers 7-4
Based Addressing 7-5
Standard Memory Models 7-5
Selecting Memory Models 7-6
Selecting from PWB 7-7
Selecting from the Command Line 7-7
Limitations on Code Size and Data Size 7-8
How to Use The Huge Memory Model 7-8
Null Pointer Sizing Issues 7-9
Mixing Memory Models 7-11
Pointer Problems 7-13
Pointer Conversions 7-15
Rules for Declaring Near, Far, Huge, and Based Variables 7-17
Rules for Declaring Near and Far Functions 7-18

Customizing Memory Models 7-20
Sizing Code Pointers 7-22
Sizing Data Pointers 7-22
Setting Up Segments 7-23
Option /Ad 7-23
Option /Au 7-24
Option /Aw 7-24
Library Support for Customized Memory Models 7-25
Setting the Data Threshold 7-26
Naming Modules and Segments 7-27
Specifying Code and Data Segments 7-29
Using Based Variables 7-30
Advantages of Based Pointers 7-30
Based Pointer Keywords 7-31
Declaring Based Variables 7-32
Variables and Pointers Based on a Segment Constant 7-33
Pointers Based on a Segment Variable 7-34
Pointers Based on a Pointer 7-36
Pointers Based on Void 7-38
Pointers Based on a Self Segment 7-38

Section 8. Using the In-Line Assembler

Advantages of In-Line Assembly 8-2
The _asm Keyword 8-2
Using Assembly Language in _asm Blocks 8-4
Instruction Set 8-4
Expressions 8-4
Data Directives and Operators 8-5
EVEN and ALIGN Directives 8-5
Macros 8-5
Segment References 8-5
Type and Variable Sizes 8-6
Comments 8-6
The _emit Pseudoinstruction 8-7
Debugging and Listings 8-7
Using C in _asm Blocks 8-8
Using Operators 8-9
Using C Symbols 8-9
Accessing C Data 8-11
Writing Functions 8-12

Using and Preserving Registers 8-14
Jumping to Labels 8-15
Calling C Functions 8-17
Defining _asm Blocks as C Macros 8-18
Optimizing 8-20

Section 9. Controlling Floating-Point Math Operations

Declaring Floating-Point Types 9-1
Declaring Variables as Floating-Point Types 9-2
Declaring Functions that Return Floating-Point Types 9-4
C Run-Time Library Support of Type long double 9-5
Summary of Math Packages 9-5
Emulator Package 9-5
Math Coprocessor Package 9-6
Alternate Math Package 9-6
Compiling and Linking Floating-Point Options 9-7
In-Line Emulator Option (/FPi) 9-10
In-Line Math Coprocessor Instructions Option (/FPi87) .. 9-11
Calls to Emulator Option (/FPc) 9-11
Calls to Math Coprocessor Option (/FPc87) 9-12
Use Alternate Math Option (/FPa) 9-13
Library Considerations for Floating-Point Options 9-13
In-Line Instructions or Calls 9-13
Compatibility between Floating-Point Options 9-14
Using the NO87 Environment Variable 9-15

Section 10. Managing Development Projects with NMAKE

How NMAKE Works 10-2
NMAKE Command Form 10-3
NMAKE Description Files 10-4
Description Blocks 10-4
Escape Sequences 10-7
Wildcards 10-7
Command Modifiers 10-8
Using Control Characters as Literals 10-8
Listing Targets in Multiple Description Blocks 10-10

Comments 10-11
Macros 10-11
User-Defined Macros 10-12
Invoking Macros 10-13
Predefined Macros 10-14
Substitution within Macros 10-17
Inherited Macros 10-19
Precedence among Macro Definitions 10-19
Inference Rules 10-20
User-Defined Inference Rules 10-20
Extension Search Paths 10-21
Predefined Inference Rules 10-22
Precedence among Inference Rules 10-22
Directives 10-23
Pseudotargets 10-26
Command Line Options 10-28
In-Line Files 10-30
NMAKE Operations Sequence 10-31
Differences between NMAKE and MAKE 10-33

Section 11. Creating Help Files with HelpMake

Help Database Content and Format 11-1
Help File Contents 11-2
Help File Formats 11-3
QuickHelp Format 11-4
Minimally Formatted ASCII 11-4
Unformatted ASCII 11-4
Rich Text Format 11-4
Using the HelpMake Command 11-5
HelpMake Options 11-6
Encoding Options 11-7
Decoding Options 11-11
Modifying a Help Database 11-13
Help Text Conventions 11-15
Help Text File Structure 11-15
Local Contexts 11-16
Context Prefixes 11-17
Hyperlinks 11-19

Using Help Database Formats 11-21
QuickHelp Format 11-22
QuickHelp Dot Commands 11-23
QuickHelp Formatting Flags 11-26
QuickHelp Cross References 11-27
Minimally Formatted ASCII Format 11-30
Rich Text Format (RTF) 11-31
Advisor Help Library 11-34

Section 12. Customizing the Programmer's WorkBench

Setting Switches 12-2
Editing from the Options Menu 12-2
Editing TOOLS.INI 12-3
Assigning Keystrokes 12-4
Changing Keystrokes 12-4
Assigning Multiple Keystrokes to One Function 12-4
Disabling Keystrokes 12-5
Limitations 12-5
Writing Macros 12-6
Macro Syntax 12-6
Literal Text 12-6
Definitions 12-7
Passing Arguments to Functions 12-7
Argument Syntax 12-8
Macros Invoking Macros 12-8
Macro Responses 12-8
Macro Arguments 12-9
Macro Conditionals 12-10
Temporary Macros 12-11
Recording Macros 12-12
Writing and Building C Extensions 12-13
Sample Extension 12-14
Writing Mandatory Extension Elements 12-16
Function Prototypes 12-16
Command Table 12-16
Switch Table 12-18
Initializing Function 12-20

Writing Extension Functions 12-20
Receiving Parameters 12-21
Using PWB Functions to Get a File Handle 12-23
Reading a Line from the File 12-24
Writing a Line to the File 12-24
Summary of PWB Functions 12-24
Building, Using, and Debugging Extensions 12-28
Compiling Extensions 12-29
Linking Extensions 12-31
Loading Extensions 12-32
Assigning Keystrokes to Extension Functions 12-33
Debugging Extensions 12-35

Section 13. Debugging C Programs with CodeView

Preparing a Debug Build from the Command Line 13-2
Compiling and Automatic Linking with MLINK 13-4
Compiling and Linking with MCBIND 13-4
Starting the CodeView Debugger 13-6
Understanding the CodeView Debugger Windows 13-8
Adjusting Windows 13-11
Overview of Debugging Techniques 13-11
Controlling Execution 13-12
Continuous Execution 13-12
Selecting and Editing Breakpoint Lines 13-13
Setting Breakpoint Values 13-15
Using Breakpoints 13-17
Single Stepping 13-18
Viewing and Modifying Program Data 13-19
Displaying Variables in the Watch Window 13-19
Displaying Expressions in the Watch Window 13-20
Displaying Arrays and Structures 13-21
Displaying Array Elements Dynamically 13-23
Using Quick Watch 13-24
Displaying Memory 13-25
Displaying Variables with a Live Expression 13-26
Displaying Processor Registers 13-27
Modifying Values of Variables, Registers, and Memory . 13-27
Memory Window 13-28
Register Window 13-28

Replaying a Debug Session 13-28
Advanced CodeView Debugger Techniques 13-29
Setting Command-Line Arguments 13-29
Multiple Source Windows 13-30
Calling Functions 13-30
Checking for Undefined Pointers 13-30
Printing Selected Items 13-31
Handling Register Variables 13-32
Redirecting CodeView Input and Output 13-32
Customizing the CodeView Debugger from TOOLS.INI . 13-33

Section 14. Programming with Mixed Languages

Making Mixed-Language Calls 14-1
Library Conflicts 14-3
Language Convention Requirements 14-4
Naming Convention Requirements 14-4
Calling Convention Requirement 14-7
Effects of Calling Conventions 14-8
Parameter-Passing Requirements 14-9
Compiling and Linking 14-11
Compiling with Correct Memory Models 14-11
C Calls to High-Level Languages 14-12
Executing Mixed-Language Calls 14-12
Using the _plm, _fortran, and _pascal Keywords 14-13
C Calls to FORTRAN 14-15
Calling a FORTRAN Subroutine from C 14-15
Calling a FORTRAN Function from C 14-17
C Calls to Pascal and PL/M 14-18
Calling a Pascal or PL/M Procedure from C 14-19
Calling a Pascal or PL/M Function from C 14-20
C Calls to Assembly Language 14-21
Writing the Assembly-Language Procedure 14-22
Setting Up the Procedure 14-23
Entering the Procedure 14-24
Allocating Local Data 14-24
Preserving Register Values 14-25
Accessing Parameters 14-27
Returning a Value 14-30
Exiting the Procedure 14-31

Handling Data in Mixed-Language Programming 14-32
Default Naming and Calling Conventions 14-33
Pascal Conventions 14-33
Numeric Data Representation 14-33
Strings 14-35
C String Format 14-35
FORTRAN String Format 14-36
Pascal String Format 14-36
Array Declaration and Indexing 14-38
Structures, Records, and User-Defined Types 14-39
External Data 14-40
Pointers and Address Variables 14-41
Common Blocks 14-42
Passing the Address of a Common Block 14-42
Accessing Common Blocks Directly 14-43
Using a Varying Number of Parameters 14-43

Section 15. Writing Portable Programs

Hardware Assumptions 15-2
Size of Basic Data Types 15-2
Type char 15-3
Type int and Type short int 15-3
Type float, Type double, and Type long double 15-5
Storage Order and Alignment 15-7
Structure Order and Alignment 15-8
Union Order and Alignment 15-10
Byte Order in a Word 15-10
Storage Alignment 15-12
Reading and Writing Structures 15-12
Bit Fields in Structures 15-12
Order of Assignment 15-13
Size and Alignment of Bit Fields 15-13
CTOS Microsoft C Specific 15-14
Processor Arithmetic Mode 15-14
Pointers 15-15
Casting Pointers 15-16
Pointer Size 15-16
Pointer Subtraction 15-16
Null Pointers 15-17
CTOS Microsoft C Specific 15-18

Address Space 15-19
Character Set 15-20
Character Classification 15-20
Case Translation 15-21
Assumptions about the Compiler 15-22
Sign Extension 15-22
Promotion from Shorter Types 15-22
Bitwise Right-Shift Operations 15-24
Length and Case of Identifiers 15-25
Register Variables 15-26
CTOS Microsoft C Specific 15-26
Functions with a Variable Number of Arguments 15-27
Evaluation Order 15-27
Function and Macro Arguments with Side Effects 15-28
Environment Differences 15-30
Portability of Data Files 15-30
Portability Concerns Specific to CTOS Microsoft C 15-31
CTOS Microsoft C Byte Ordering 15-31

Section 16. Building CTOS III Applications

Creating Multithread CTOS III Applications 16-1
Multithread Programs 16-1
Library Support 16-2
The Multithread C Library LLIBCMT.LIB 16-3
The Multithread Library Compile Option (/MT) 16-4
Include Files 16-5
C Run-Time Library Functions for Thread Control 16-5
The _beginthread Function 16-6
The _endthread Function 16-7
Writing a Multithread Program 16-7
Sharing Common Resources 16-7
Thread Stacks 16-8
Dynamic Linking with CTOS III 16-8
Overview of Dynamic Linking 16-9
Load-Time and Run-Time Linking 16-9
Application Programs and DLLs 16-10
DLLs and CTOS Microsoft C Run-Time Libraries 16-10
Standalone Dynamic-Link Libraries 16-11
DLLs without C Run-Time Library Functions 16-12

Designing and Writing DLLs 16-13
Floating-Point Math Requirements 16-13
Initialization and Termination Requirements 16-14
Initialization 16-14
Termination 16-16
Making the DLL Re-Entrant 16-16
Global Versus Instance Data 16-17
Using CTOS Microsoft C Keywords 16-18
The _export Keyword 16-18
The _loadds Keyword 16-19
Compile Options for Dynamic-Link Libraries 16-19
Compile without Linking (/c) 16-19
Large Memory Model with Separate Stack (/ALw) 16-19
Remove Stack Probes (/Gs) 16-20
Specify 80286 Code (/G2) 16-20
Link C Run-Time into Stand-Along DLL (/ML) ... 16-20
Link Executable or DLL with C Run-Time DLL (/MD) 16-20
Building DLLs with CTOS Microsoft C 16-21
DLLs with Static C Run-Time Library Functions ... 16-21
Building the DLL 16-21
Building Programs that Call the DLL 16-25
Additional Information 16-27

Appendix A. Migrating to CTOS Microsoft C

Program Changes A-2
C Functions A-2
Header Files A-6
Notes A-6
Using ctos.lib.h A-7
Header File Conversions A-7
Memory Models A-11
Language Keywords A-12
Pragmas A-13
Mixed-Language Programming A-14
Coding Conventions A-14
Function Prototypes A-14
Initialization A-15
Function and Macro Names A-15

Compiler Options A-15
Predefined Macros A-20
COMDEF OMF Versus PUBDEF Records A-20
DS Allocation A-21
Overlays A-21
C Library Functions A-22
Math Library Functions A-30

Appendix B. Command Reference

CL B-1
Environment Variables Recognized B-2
Command Line Options B-2
CodeView B-10
Summary B-10
Syntax B-10
Environment Variables Recognized B-10
Command Line Options B-11
CodeView Commands B-12
Dialog Commands B-14
Size Specifiers B-18
Format Specifiers B-18
Environment Service B-19
Summary B-19
Command B-20
Variable, Examples, and Commands B-21
Variable Descriptions B-22
HELPMAKE B-23
Summary B-23
Environment Variables Recognized B-23
Command Line Options B-23
MLIB B-28
Summary B-28
Environment Variables Recognized B-28
Command Line Options B-28
MLINK B-29
Summary B-29
Syntax B-29
Environment Variables Recognized B-30
Command Line Options B-30

NMAKE B-38
Summary B-38
Syntax B-38
Environment Variables Recognized B-38
Command Line Options B-38
Programmer's WorkBench B-40
Summary B-40
Environment Variables Recognized B-40
Command Line Options B-40
PWBRMake B-41
Summary B-41
Environment Variables Recognized B-41
Command Line Options B-42
QuickHelp B-43
Environment Variables Recognized B-43
Command Line Options B-43

Appendix C. Implementation-Defined Behavior

Translation C-2
Diagnostics C-2
Environment C-2
Arguments to main C-3
Interactive Devices C-3
Identifiers C-4
Significant Characters without External Linkage C-4
Uppercase and Lowercase C-4
Characters C-5
Multibyte Characters C-5
Bits per Character C-5
Character Sets C-5
Unrepresented Character Constants C-6
Wide Characters C-6
Converting Multibyte Characters C-7
Range of char Values C-7
Integers C-7
Range of Integer Values C-8
Demotion of Integers C-8
Signed Bitwise Operations C-9
Remainders C-9
Right Shifts C-9

Floating-Point Math C-10
Values C-10
Casting Integers to Floating-Point Values C-10
Truncation of Floating-Point Values C-11
Arrays and Pointers C-11
Largest Array Size C-11
Casting Pointers C-11
Pointer Subtraction C-12
Registers C-12
Availability of Registers C-12
Structures, Unions, Enumerations, and Bit Fields C-13
Improper Access to a Union C-13
Sign of Bit Fields C-13
Storage of Bit Fields C-14
Alignment of Bit Fields C-14
The Enum Type C-15
Qualifiers C-15
Access to Volatile Objects C-15
Declarators C-15
Maximum Number C-15
Statements C-15
Limits on Switch Statements C-15
Preprocessing Directives C-16
Character Constants and Conditional Inclusion C-16
Including Bracketted File Names C-16
Including Quoted File Names C-17
Character Sequences C-18
Pragmas C-18
Default Date and Time C-19
Library Functions C-19
NULL Macro C-20
Diagnostic Printed by the assert Function C-21
Character Testing C-21
Domain Errors C-22
Underflow of Floating-Point Values C-22
fmod Function C-22
Terminating Newline Characters C-22
Blank Lines C-23
Null Characters C-23
File Position in Append Mode C-23
Truncation of Text Files C-23

File Buffering C-23
Zero-Length Files C-24
File Names C-24
File Access Limits C-24
Deleting Open Files C-24
Renaming with a Name that Exists C-25
Printing Pointer Values C-25
Reading Pointer Values C-25
Reading Ranges C-25
File Position Errors C-26
Messages Generated by the perror Function C-26
Allocating Zero Memory C-27
abort Function C-27
atexit Function C-27
Environment Names C-28
system Function C-28
strerror Function C-28
Time Zone C-30
Clock Function C-30

Appendix D. Compiler Limits and Numerical Ranges

Compiler Limits D-2
Numerical Ranges D-4

Appendix E. printf/scanf Format Specifiers E-1

Appendix F. Re-entrant Run-Time Library Functions F-1

Appendix G. Sample NMAKE Description Files

Help File G-2
Library Version String G-4
Programmer's WorkBench Extension G-6
Run File G-12
Library G-21
Run Files, System Service, and Libraries G-30

Glossary Glossary-1

Index Index-1

Figures

1-1. Building Programs in One Step 1-2
1-2. Building Programs in Two Steps 1-3
1-3. Building Programs with a Makefile 1-4
1-4. Building and Debugging Programs from PWB 1-6
1-5. Task Management Service 1-8
1-6. Relationships between Help Utilities 1-10

3-1. Parts of a PWB Screen. 3-5
3-2. Dialog Box for the Find Command 3-16
3-3. Define Mark Dialog Box 3-22
3-4. Setting and Saving Build Options 3-32
3-5. Creating a Makefile 3-33
3-6. Building Programs 3-34
3-7. Dialog Box for the Save As Command 3-35
3-8. C Compiler Options Menu 3-36
3-9. C Compiler Debug Options Menu 3-37
3-10. C Compiler Release Options Menu 3-38
3-11. Link Options Menu 3-39
3-12. Link Debug Options Menu 3-40
3-13. Link Release Options Menu 3-41
3-14. Build Options Dialog Box 3-42
3-15. Dialog Box for the Set Program Command 3-43
3-16. Makefile Dialog Box 3-44
3-17. Edit Program List Command Form 3-44
3-18. Set Initial Build Options Dialog box 3-46
3-19. Browse Options Dialog Box 3-56
3-20. Browse Menu 3-58
3-21. Browser Display of Functions 3-61
3-22. CodeView Options Menu 3-65
3-23. CodeView Startup Screen 3-66
3-24. CodeView Screen with Watch and Register Windows 3-67
3-25. CodeView Screen with Expanded Arrays 3-68

4-1. QuickHelp Initial Screen 4-4
4-2. Global Contents Help Screen 4-5
4-3. Global Index Help Screen 4-6
4-4. File Commands 4-8
4-5. View Commands 4-9
4-6. Categories Commands 4-10
4-7. References Commands 4-11
4-8. Paste Commands 4-12
4-9. Options Commands 4-13
4-10. Dialog Box for the Search Command 4-15
4-11. Sample Index for Keywords 4-22

5-1. MCBIND Command Form 5-15
5-2. Sample MCBIND Link 5-20
5-3. Near Heap Memory Organization Options 5-29

7-1. Anatomy of Small-Model Program 7-3

10-1. Typical Description Block 10-5

12-1. Sample CL Command Form for Building Extensions 12-29
12-2. Sample CL Command Form for Compiling Extensions 12-30
12-3. Sample MCBIND Command Form for Building Extensions 12-31

13-1. Sample CL Command with Linking for a CodeView Build 13-4
13-2. Sample CL Command without Linking for a CodeView Build 13-4
13-3. Sample MCBIND Command for CodeView Build 13-5
13-4. Sample CodeView Startup Screen 13-9
13-5. Edit Breakpoints Dialog Box 13-14
13-6. Set Breakpoint Dialog Box 13-16
13-7. Add Watch Dialog Box 13-19
13-8. Quick Watch Dialog Box 13-24
13-9. Memory Window Options 13-25
13-10. Print Dialog Box 13-31

14-1. Mixed-Language Call 14-2
14-2. Naming Convention 14-6
14-3. C Stack Frame 14-28
14-4. C String Format 14-35
14-5. FORTRAN String Format 14-36

15-1. Data Alignment in Bit Fields 15-14

16-1. DLL and Program with Statically Linked C Run-Time Functions 16-12

Tables

2-1. Software Selection 2-6
2-2. Addtional Software Selection 2-8

3-1. Parts of a PWB Screen 3-6
3-2. Command Overview 3-9
3-3. PWB Dialog Box Features 3-16
3-4. Browse Commands 3-59

4-1. Hyperlink Commands 4-17
4-2. Ways to Display Unmarked Hyperlink Topics 4-18

5-1. Compiler Options Used for Overlays 5-4
5-2. CL Compiler Options Incompatible with Overlays 5-5
5-3. MLINK Command Fields 5-8
5-4. MLINK Defaults Correlated to MCBIND Fields 5-13
5-5. Correlating MCBIND and MLINK Options 5-16
5-6. Correlating CTOS Linker Config File and MLINK Options 5-18
5-7. Startup Object Modules 5-22
5-8. Flavor Object Modules 5-22
5-9. Standard Libraries 5-23
5-10. Combined Libraries 5-26
5-11. Unsupported Library Functions 5-43

6-1. Processor Compatibility 6-22
6-2. Processor Clock Cycles for Calling Sequence 6-27
6-3. Register Candidates 6-31

7-1. Memory Models 7-6
7-2. Memory Model Options and Libraries 7-7
7-3. Addressing Declared with Microsoft Keywords 7-12
7-4. Start-Up Routines for Customized Memory Models 7-26
7-5. Segment-Naming Conventions 7-28
7-6. Keywords 7-32

9-1. Floating-Point Types 9-2
9-2. Lengths of Exponents and Mantissas 9-2
9-3. Range of Floating-Point Types 9-3
9-4. Summary of Floating-Point Compiler Options 9-8
9-5. Floating-Point Options and Libraries 9-9

10-1. Command Modifiers 10-8
10-2. Predefined Macros 10-14
10-3. Predefined Inference Rules 10-22
10-4. Directives 10-23
10-5. Directive Operators 10-25
10-6. Pseudotargets 10-27
10-7. NMAKE Options 10-28

11-1. Options for Encoding 11-7
11-2. Options for Decoding 11-11
11-3. Standard h. Contexts 11-17
11-4. Microsoft Product Context Prefixes 11-19
11-5. QuickHelp Dot Commands 11-24
11-6. Formatting Flags 11-26

12-1. Callable PWB Functions 12-25

13-1. List of CodeView Windows 13-8

14-1. Language Equivalents for Routine Calls 14-3
14-2. Methods for Passing Parameters 14-10
14-3. Parameter-Passing Defaults 14-11
14-4. Register Conventions for Simple Return Values 14-30
14-5. Default Naming and Calling Conventions 14-33
14-6. Equivalent Numeric Data Types 14-34
14-7. Equivalent Array Declarations 14-39

15-1. Size of Basic Types in CTOS Microsoft C 15-3
15-2. Size of Generic Pointers 15-18
15-3. Default Pointer Sizes 15-19
15-4. Byte Ordering for Short Types 15-32
15-5. Byte Ordering for Long Types 15-32

16-1. Multithread Options 16-4
16-2. Stand-Alone DLL Files 16-21
16-3. Linking Files Needed 16-26

A-1. Documented BTOS C Functions Without Substitutes A-3
A-2. Documented BTOS C Functions With Substitutes A-4
A-3. Undocumented BTOS C Functions With Work-arounds A-5
A-4. BTOS C and CTOS Microsoft C Header Files A-8
A-5. BTOS C and CTOS Microsoft C Extended Keywords A-13
A-6. BTOS C and CTOS Microsoft C Compiler Options A-16
A-7. BTOS C Function Status in CTOS Microsoft C A-22
A-8. BTOS C Math Function Locations in CTOS Microsoft C A-30

C-2. Source and Execution Character Sets C-6

D-1. Limits Imposed by the C Compiler D-2
D-2. Program Limits at Run Time D-3
D-3. Data Ranges Defined in LIMITS.H D-4
D-4. Numerical Values Defined in FLOAT.H D-6

E-1. printf and scanf Format Specifiers E-1

This Page is Blank

Do Not Print

Section 1
Overview

CTOS Microsoft C is a C-language development system that provides professional developers with sophisticated tools to create CTOS applications. CTOS Microsoft C is ported from the Microsoft C Professional Development System, version 6.0.

With CTOS Microsoft C, you can build programs four ways:

- Compile and link in one step from the Executive
- Compile and link in two steps from the Executive
- Execute a makefile from the Executive
- Compile and link from the Programmer's WorkBench

CTOS Microsoft C includes supporting utilities to help you build programs including online Help, a librarian, a browser, and a debugger. This overview briefly describes the following:

- Compiler and Linker
- NMAKE
- The Programmer's WorkBench
- The CodeView Debugger
- A Source Browser
- Task Management Service
- Online Help Utilities
- Librarian
- Environment Service
- New Features

Compiler and Linker

To build a run file with Executive commands, you compile your source code with the CL command, and link with one of two interfaces to the CTOS Linker: MLINK or MCBIND. Unless you disable automatic linking, CL chains to MLINK as shown in Figure 1-1. MLINK creates your run file automatically. You do not have to complete a linker command form.

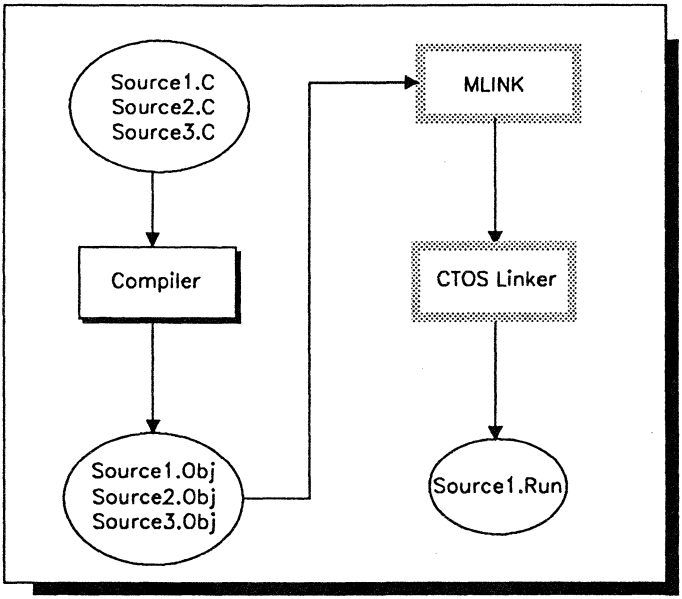


Figure 1-1. Building Programs in One Step

If you want to use a CTOS command form for linking, use MCBIND, the other interface to the CTOS Linker. To use MCBIND, you must disable automatic linking when compiling. Once CL creates your object files, you specify them in the MCBIND command form along with startup object modules, libraries, and other parameters. If you wish, you can also disable automatic linking and use MLINK as a standalone interface to the CTOS Linker.

Figure 1-2 illustrates compiling and linking in separate steps. Section 5, Building Applications, provides the detailed information you need to compile and link with CL, MLINK, and MCBIND.

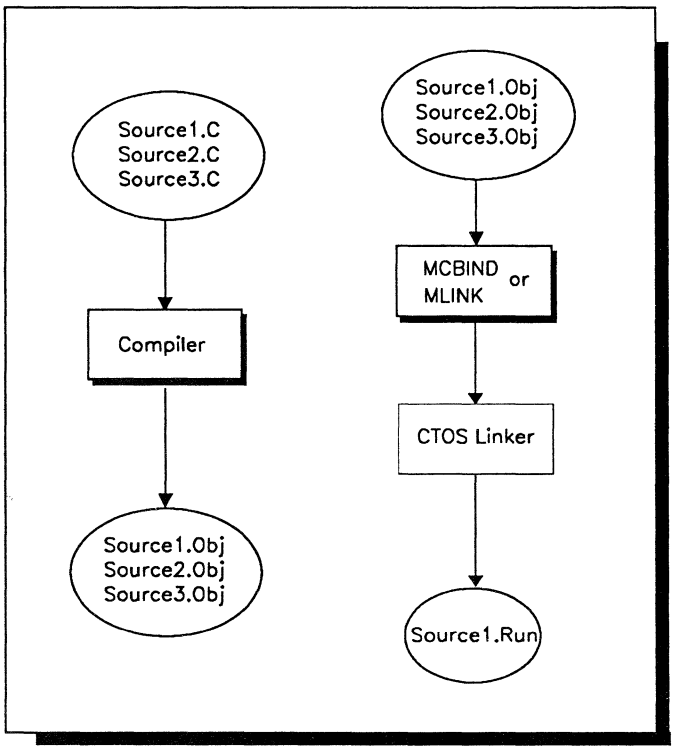


Figure 1-2. Building Programs in Two Steps

NMAKE

Developers who create large applications most often control their build process with makefiles and NMAKE. In makefiles (also called description files), you list the commands, source files, and object files that create your run files as shown in Figure 1-3. You then execute NMAKE with your makefile instead of executing separate commands. By listing build information this way, you ensure consistent builds.

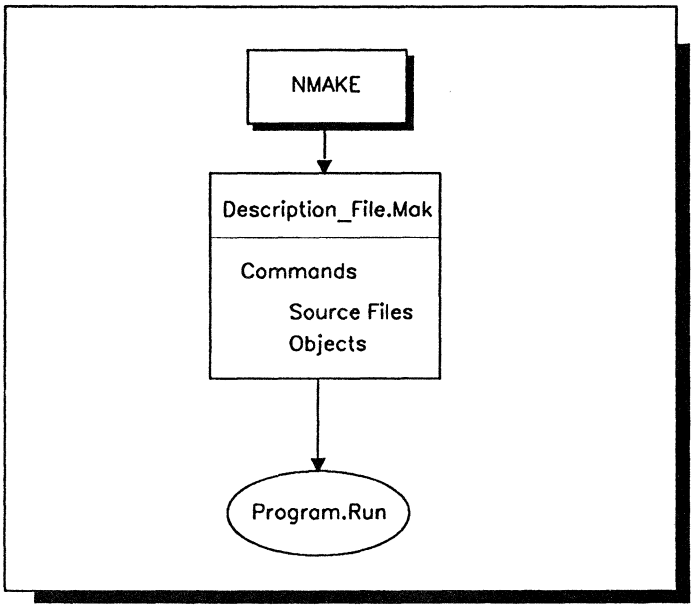


Figure 1-3. Building Programs with a Makefile

NMAKE rebuilds out-of-date files only. This means that if you list 100 source files in your makefile and change one since your previous build, NMAKE first rebuilds the object for the changed file, and then rebuilds any objects that use the changed file.

To simplify text entry in makefiles, NMAKE supports user-defined macros and recognizes a set of predefined macros. NMAKE also recognizes IF/ELSE directives, C-language operators (%, !=, and so on), and inference rules. For sample makefiles, refer to Appendix G, which contains makefiles used to create the CTOS Microsoft C development system.

The Programmer's WorkBench

The Programmer's WorkBench (PWB) offers an integrated environment from which you can perform all programming tasks. You can create and edit source files, and add them to a PWB-type makefile. You can also compile, link, and execute run files.

Additionally, you can view compiler errors, and have the Editor place the cursor on source lines that contain errors. To diagnose problems, you can use the Source Browser and the CodeView Debugger.

PWB also allows you to define compiler and linker options for building two versions of your run files: debug and release. During initial stages of development, you can build debug versions: unoptimized run files that also contain information used by one or both debuggers. During final stages of development, you can build release versions: fully optimized run files without debugging information.

Figure 1-4 illustrates the automatic processing that occurs when building programs from PWB, and how PWB integrates the debuggers.

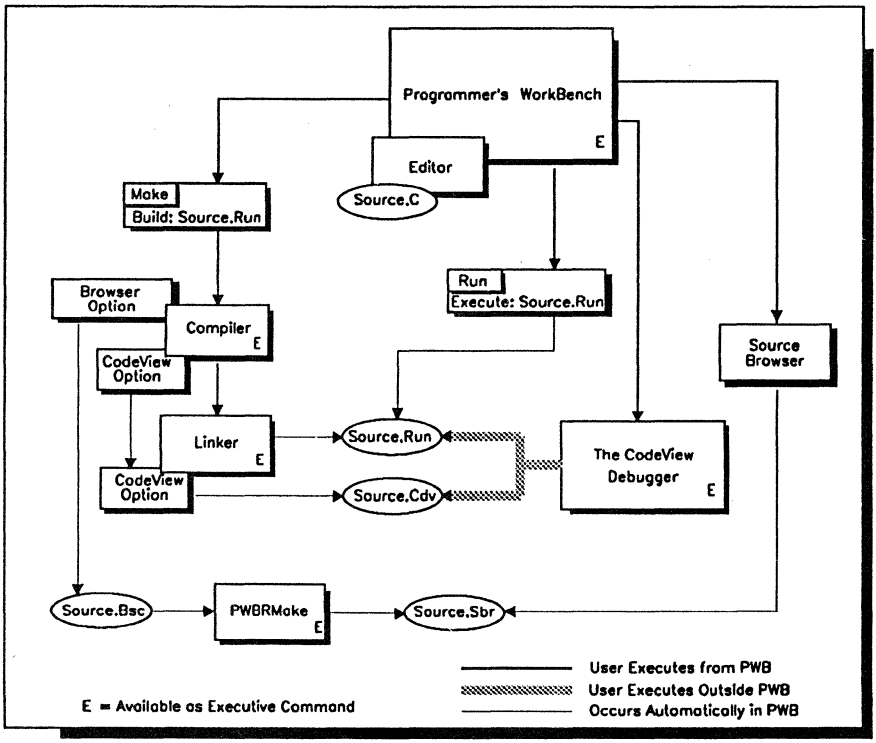


Figure 1-4. Building and Debugging Programs from PWB

The CodeView Debugger

The CodeView debugger, which you can run from PWB and as a standalone utility, is a symbolic debugger. Briefly, the CodeView debugger lets you watch your code as it executes line by line, watch variables as they change value, and display and modify register values. For the CodeView debugger to work at a full symbolic level, you must compile and link with CodeView options.

The Browser

The Browser is available from PWB only. With the Browser, you can search multiple files to list function references, display lines that contain these references, list relationships among program symbols, and view a call tree. For example, you can generate a list of all functions used in your code, and display lines that call a function in the list by clicking your mouse on the desired function. For the Browser to work, you must compile with an option for Browser output. If you compile from the Executive command line instead of PWB, you must also use PWBRMAKE.

Task Management Service

Task Management Service is the CTOS system service used by the Programmer's WorkBench and CodeView debugger as shown in Figure 1-5. Both utilities require the service for background processing.

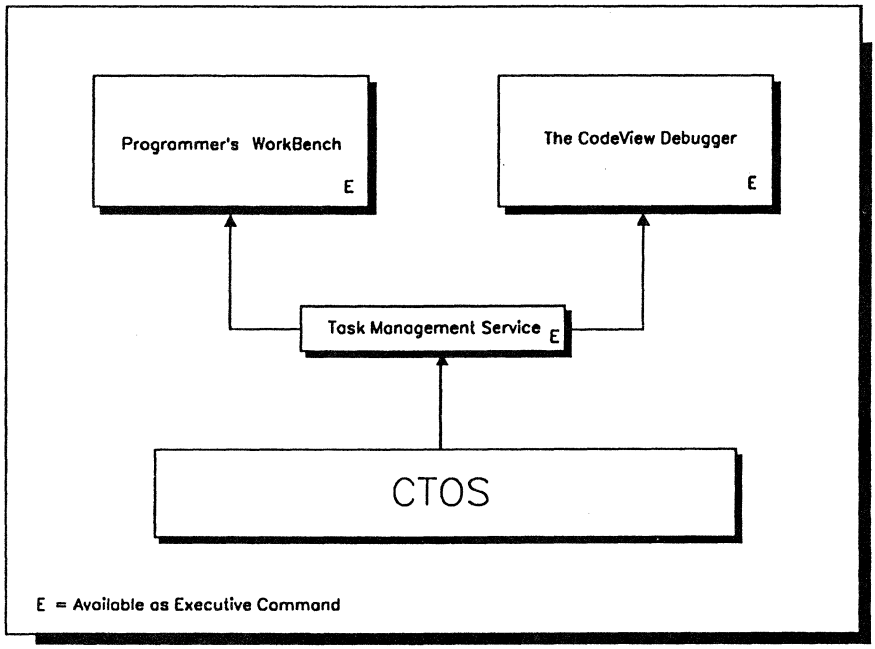


Figure 1-5 . Task Management Service

Online Help Utilities

The online Help system is an integral part of CTOS Microsoft C, and is a primary source of information about the product, including the CTOS API and CTOS structures. The system contains information not found in this manual.

The system reads a Help database composed of a series of files with the extension .Hlp. You can display these files with QuickHelp from the Executive command line, and with the Advisor from PWB and the CodeView debugger.

In addition to using Help, you can create or modify Help files with the HelpMake utility. You can use HelpMake to modify the Help files supplied with CTOS Microsoft C and customize the screens to your needs.

Figure 1-6 shows the relationships between the Help utilities.

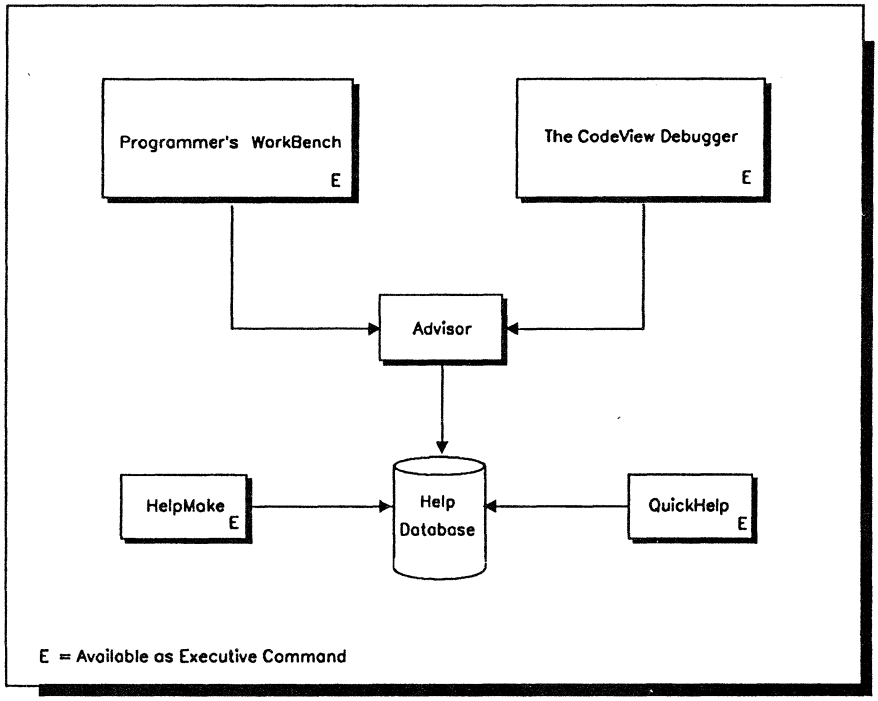


Figure 1-6. Relationships between Help Utilities

Librarian

CTOS Microsoft C includes a librarian called MLIB. You can use MLIB to create, organize, and maintain library files. Additionally, you can use it to append an object file or library to another library, delete a module from a library, and extract a module from a library and save it as an object file.

Online Help is your primary source of information about using MLIB. However, Appendix B contains command options and Section 5, Building Applications, describes how to use MLIB to create combined libraries.

Environment Service

To configure your build environment (specify search paths and so forth), you set environment variables with the Environment Service package overpacked with CTOS Microsoft 6.1. With environment variables, you can direct utilities to look in specific directories for different types of files, such as #includes, libraries, and Help files. You can also list commonly used compiler and linker options to avoid specifying them on the command line each time you use the commands.

For complete information about Environment Service, refer to the *Environment Service Administration and Programming Guide*, overpacked with CTOS Microsoft C 6.1. For a list of environment variables recognized by each utility, refer to Appendix B.

New Features

This subsection describes the major enhancements to CTOS Microsoft C 6.1.

- The Programmer's WorkBench--an integrated development environment for building CTOS applications. PWB's features include a text editor, source browser, access to extensive online Help, and a compile-link-run cycle.
- The CTOS Microsoft CodeView debugger--a symbolic debugger that allows you to run programs under debug control, and simultaneously display source code, variables, memory locations, processor registers, and more. To use it, you must successfully compile and link with CodeView options.
- Task Management Service--a CTOS system service required by the Programmer's WorkBench and CodeView debugger; optionally required by NMAKE.
- NMAKE--an additional method of makefile processing. Once targets are evaluated for updating, the "inline" method (requires support of Task Management Service) immediately executes the commands to update in a secondary partition. The previous "deferred" method of building submit files remains and is selectable with the command line switch /Y.

- MLINK--a new interface to the CTOS Linker that accepts MS-DOS-type command line options. It provides complete control over all CTOS Linker fields and configuration file arguments.
- Automatic Linking--occurs with MLINK after successful compiles. You can disable this feature.
- The _plm keyword--eliminates Envelope technology familiar to 6.0 users.
- Environment Service--provides environmental variable support and replaces 6.0's Task Manager.
- Overlay-compatible code generator--code that conforms to the requirements of the CTOS Virtual Code Segment Management facility, as enforced by the CTOS Linker.

Section 2
Installation and Configuration

This section describes how to install and configure CTOS Microsoft C.

System Requirements

The CTOS Microsoft C utilities other than the CodeView Debugger, are compatible with the following operating systems:

- BTOS II 3.0 and higher
- CTOS/VM 2.3 and higher
- CTOS II 3.0 and higher
- CTOS III
- CTOS/XE 3.0

The package is also compatible with Standard Software level 12.1. The CodeView debugger is not supported on CTOS versions older than CTOS II 3.3.

Four megabytes of RAM is recommended to use the complete package. For additional details on system requirements, refer to the Software Release Announcement included in your CTOS Microsoft C package.

Before You Install

This subsection describes pre-installation operations and decisions. If you perform these operations and make these decisions before installation, you may save time. This subsection covers these topics:

- Preliminary Operations
- Installation Decisions

Preliminary Operations

The installation procedures make two requests for information. Supplying the correct information can help you avoid an abort during installation.

- If you are installing the package on a server, power down all cluster workstations.
- Execute Volume Status to verify that you have the required disk space of 19,000 sectors for the complete system. If you do not have 19,000 free sectors, you may decline to install one or more optional utilities.

Installation Decisions

BTOS II 3.1 and higher, and CTOS/VM 2.3 and higher use submit files to install the software. You begin the submit process with the **Software Installation** or **XESoftware Installation** command.

CTOS II 3.0 and higher, CTOS III, and CTOS/XE 3.0 use Installation Manager to install the software. You begin Installation Manager with the **Installation Manager**, **Floppy Install**, **Tape Install**, or **Server Install** command. Unisys recommends this method of installation.

When using Installation Manager to install, you need to decide:

- which software and optional utilities to install and where to install them
- to accept or change defaults, including:
  - public
  - verbose
  - backup previous version
  - save copy of backup
  - save defaults in user file
  - software destination [sys],
  - user name, cm configfile name, and command file name

  You press Y or N to indicate yes or no to each default or type in the desired file name.

  For information on choosing the software, refer to Selecting Software and Directories. The following paragraphs discuss the default options.

Public

To install software on the local workstation, select **No** for a private installation.

To install software on the server, select **Yes** for a public installation. All workstations on the cluster can use publicly installed software on the cluster. Note that for CTOS Microsoft C to be installable on cluster workstations from the server, it must be installed as public software on the server.

A public installation saves time installing the software and requires fewer resources to store the software. A private installation requires more resources on each system, but requires less time for software to be loaded into memory from disk; and, in many cases, can be used on local workstations without dependency on the server.

Note: If you install Microsoft C publicly on the server and plan to use the server as a workstation, merge Cluster.cmds and the Sys.cmds on the server to ensure that users at the server have the proper forms of the commands.

Verbose

Select **Yes** to have Installation Manager display command output from the installation script on the screen.

Select **No** for a silent installation in which only a few progress messages display.

Backup Previous Version

Select **Yes** to back up the current version of a software product before the new version is installed. When you do this, the backed up version can be restored if the installation fails.

Select **No** if you do not want to back up the current version.

Save Copy of Backup

Select **Yes** if you want to save a copy of the backup. You can use this backup later to restore the previous version, if necessary.

Select **No** if you do not want to save a copy of the backup.

Save Defaults in User File

Select **Yes** to change your user file entries to the parameters you select. They become the new defaults for the Installation Manager.

Select **No** if you do not want to change your user file.

Software Destination [sys]

Enter the name of the volume that you want Microsoft C to be installed on.

User Name, CM Configfile Name, Command File Name

Enter the names of these files as Installation Manager prompts you.

Selecting Software and Directories

The installation process prompts you to select the Microsoft C software that you want to install. Once you select the sofware, you will be prompted to accept the default directory or specify your own directory. Making these decisions before you install saves time.

Table 2-1 describes the software, number of sectors required, and the defaults.

Notes: *If you are using Installation Manager, you can indicate user-specified directories only if you install privately. If you install publically or from Qic tape, you cannot indicate user-specified directories.*

If you use install with the Software Installation or XESoftware Installation commands, the primary system utilities default to <Sys>; however, you can change it to <MSC>.

Table 2-1. Software Selection

Software	OS	Sectors	Default Directory	Description
Compiler	BTOS II 3.0 and up CTOS/VM 2.3 and up CTOS II 3.0 and up CTOS III CTOS/XE 3.0	9500	MSC	Microsoft C compiler and command line utilities
PWB	CTOS II 3.3 and up CTOS III CTO/XE 3.0	1527	PWB	The integrated programming environment.
CodeView	CTOS II 3.3 and up CTOS III CTOS/XE 3.0	1019	CV	The symbolic debugger.

continued

Table 2-1. Software Selection (cont.)

Software	OS	Sectors	Default Directory	Description
Debugger3.3Sys	CTOS II 3.3 only	Overwrite	xxx	Required for CodeView ***Note:*** *Debugger3.3Sys is not part of Installation Manager's menu. You are prompted to install it during the installation process.*
Help FIles	BTOS II 3.0 and up CTOS/VM 2.3 and up CTOS II 3.0 and up CTOS III CTOS/XE 3.0	5312	MscHelp	Required to use the Help system.

Additional Software

The Environment Service and Task Management Service are overpacked with CTOS Microsoft C. If you installing from Qic tape, these products are on separate floppies. Table 2-2 describes the software.

Table 2-2. Addtional Software Selection

Sectors	Software	Description
48	Task Management Service	Required to run Programmer's WorkBench and the CodeView Debugger.
179	Environment Service	Required to use environment variables.

Installing CTOS Microsoft C

You can install CTOS Microsoft C from diskettes, tape, or your server workstation.

If you are using the submit process and are familiar with installing software, follow the prompts on the screen.

If you are using Installation Manager, each method provides a menu for you to select the software. To select/deselect items from a menu:

- To select an item, move the highlighted bar to the item and press MARK.
- To select all, press MARK on the first item, move the highlight to the last item, and press BOUND.
- To deselect, press CODE-MARK.

Once you have selected those items from the menu that meet your needs, press GO.

Installing from Diskettes

You can install CTOS Microsoft C from either 5-1/4" or 3-1/2" diskettes, using one of four commands.

To install CTOS Microsoft C from diskettes, follow these steps:

1. Insert Disk 1 at the Executive command line.
2. Type one of the following commands; then press GO.
   - **Software Installation**
   - **XESoftare Installation**
   - **Floppy Install**
   - **Install**

   The appropriate form appears.
3. Follow the prompts on the screen, using Table 2-1 as a guideline to select software.

   Installation continues until complete.
4. Reboot your system.

Note: *You can quit the installation at any time by pressing FINISH. If you are using Installation Manager, you can resume installation at the point where you quit.*

Installing from Tape

You can install CTOS Microsoft C from Qic tape if you have Standard Software 12.0.0 or greater. Qic tape installation is through Installation Manager only. The following procedure describes how to install the package as supplied on tape.

To install CTOS Microsoft C from Qic tape, follow these steps:

1. Install the tape into the tape drive.
2. At the Executive, type **Tape Install**; then, press GO.

3. Follow the prompts on the screen, using Table 2-1 as a guideline to select software.

   Installation continues until complete.

   Note: If you select Codeview and you have CTOS II 3.3.x, you will be prompted to install Debugger3.3.Sys.

4. When installation is complete, remove the tape from tape drive.

Installing from Your Server Workstation

If you are clustered to a server, you can install CTOS Microsoft C onto your workstation from your server workstation. This method is much quicker than installing from the diskettes onto your workstation. The following procedure describes how to install the package from the server.

To install CTOS Microsoft C from the server, follow these steps:

1. At the Executive, type **Server Install**; then, press GO.
2. Follow the prompts on the screen, using Table 2-1 as a guideline to select default directories.

   Installation continues until complete.

Installing Environment Service and Task Management Service

The Environment Service (EVS) and Task Management Service (TMS) are on separate diskettes. You install them through the Installation Manager. The following procedure describes how to install these products.

To install either EVS or TMS, follow these steps:

1. Insert the first floppy diskette into the floppy drive.
2. Type one of the following commands, then press RETURN.
   - **Floppy Install**
   - **Install**

   The appropriate form appears.
3. Follow the prompts on the screen

   Installation continues until complete.
4. Reboot your system.

Configuring Your System

This subsection describes configuration information you should use to get up and running quickly, and covers these topics:

- Installing Task Management Service
- Setting up Environment Service
- The CodeView Debugger
- CTOS Libraries
- Video Note
- Increasing the Maximum Number of Open Files

Installing Task Management Service

Task Management Service (TMS) supports PWB's and the CodeView debugger's background interaction with CTOS. For example, PWB uses TMS to run the compiler as a background task.

To install the service, use this command:

```
Install Task Management Service
```

Alternatively, you can add this line to SysInit.Jcl:

```
$RUN [Vol]<Dir>TMS.Run
```

If you use Batch Manager, specifically Batch BS Service, you must install Batch BS Service before TMS.

If you use Context Manager, you must install TMS before Context Manager.

The files TMS.Run and TMSNull.Run must be in the same directory.

Setting up Environment Service

Environment Service allows you to set variables used by commands. With this utility, you can eliminate having to specify command path statements for include, library, and Help files, and set commonly used command options.

Environment Service reads information specified in EnvironmentConfig.Sys. CTOS Microsoft C supplies a sample config file called `SAMPLE>EnvironmentConfig.Sys` that looks like this:

```
:INCLUDE:[Sys]<MscIncludes>
:CL:/nologo
:HELPFILES:[Sys]<MscHelp>
:LIB:[Sys]<MscLib>
:TMP:[Scr]<$>
```

To install the service, use this command:

```
INSTALL ENVIRONMENT SERVICE
    [Configuration file]
    [Service name]
    [Case sensitive]
```

Alternatively, you can add this line to SysInit.Jcl:

```
$RUN [Sys]<Sys>EVS.Run
```

If you use Context Manager, you must install Environment Service before Context Manager.

Sample EnvironmentConfig.Sys

To show how Environment Service uses EnvironmentConfig.Sys, the following example configures your system to build a run file with one compile command, and creates a file required by the CodeView debugger. To use this example, first copy SAMPLE>EnvironmentConfig.Sys to [Sys]<Sys>, rename it to EnvironmentConfig.Sys, edit and save it to match the example. Be sure to supply your own directories if they differ.

```
:CL: /AL /Od /Zi
:INCLUDE: [Sys]<MscIncludes>
:PATH: [Sys]<Your_Source_Dir>; [sys]<MSCHelp>
:HELPFILES: [Sys]<MscHelp>
:QH: [Sys]<MscHelp>
:LIB: [Sys]<MscLib>
:TMP: [Scr]<$>
```

You can install this file with the following command:

```
INSTALL ENVIRONMENT SERVICE
    [Configuration file]  <Sys>EnvironmentConfig.Sys
    [Service name]
    [Case sensitive]
```

After installing the above configuration, you can now build a run file by compiling from your source directory with this command:

```
CL
   [Args]     Source.c
   [     ]
```

Everything else is automatic. The compiler and linker build with the large model (/AL), create an unoptimized run file (/Od), know where to find include and library files, and create the files necessary to use the CodeView debugger (/Zi). For details, refer to Section 5, Building Applications

Note: *The :PATH: variable does not affect CL, so you have to compile from your source directory, or specify your directory on the command line.*

For details on all variables recognized with CTOS Microsoft C utilities, refer to Appendix B. For complete information about Environment Service, refer to your *Environment Service Administration and Programming Guide.*

The CodeView Debugger

The CodeView debugger requires the operating system CTOS II 3.3 or greater, and Task Management Service. Additionally, to run on CTOS II 3.3.x only, the CodeView debugger requires the special version of Debugger3.3.Sys, available during installation. To run CodeView on CTOS III, do not accept Debugger3.3.Sys during installation.

Workstations require at least 4 mb of memory to run CodeView.

CTOS Libraries

It is assumed that users of CTOS Microsoft C are CTOS-development professionals, and work with two CTOS development libraries located in [Sys]<Sys>:

CTOS.Lib
CTOSToolKit.Lib

You cannot build run files without these libraries, and CTOS Microsoft C does not package them.

Video Note

To allow QuickHelp and PWB to work properly with VGA on CTOS II 3.3.x or on CTOS III, the following line is required in [Sys]<Sys>Config.Sys:

```
:EVBackgroundOff:No
```

Increasing the Maximum Number of Open Files

CTOS Microsoft C allows you to increase the maximum number of files that may be open for I/O (the default number is 20). To accomplish this change, you need the CTOS Microsoft C Library Source package, available from your designated product support (or have your local Unisys marketing manager contact the CTOS Marketing organization in San Jose).

The library source files that must be modified are:

<CRT_STARTUP>Crt0dat.Asm
<LIBCEXT>ApiFile.c
<CRT_STDIO>_file.cc

Increasing File Handles

To increase the number of file handles, first edit the start-up source file <CRT_STARTUP>Crt0dat.Asm, and find the line:

```
_NFILE_ = 20
```

Change 20 to the desired maximum. For example, to increase the maximum number of available file handles to 40, change the line to:

```
_NFILE_ = 40
```

Next, edit <LIBCEXT>ApiFile.c , and find the line :

```
#define  MAXHANDLES  20
```

Change 20 to the to the value of _NFILE_. For example, if you changed _NFILE_ to 40, change the line to:

```
#define  MAXHANDLES  40
```

Increasing Streams

Increasing the number of file handles allows you to use low-level (or block) I/O functions, such as open and read, with more files, but does not increase the number of streams (that is, FILE * streams). Increasing the number of streams allows you to use the stream I/O functions, such as fopen and fread, with more files.

For a streams increase, edit the source file <CRT_STDIO>_file.c, and find the line:

```
#define  _NFILE_  20
```

Change 20 to the desired maximum. For example, to increase the maximum number of streams to 40, for example, change the line to:

```
#define  _NFILE_  40
```

The number of low-level file handles must be greater than or equal to the number of stream-level files. Thus, if you increase the value of _NFILE_ in _file.c, you must also increase the value of _NFILE_ in Crt0dat.Asm.

Using the Modified Files

After you modify Crt0dat.Asm, ApiFile.c, and possibly _file.c, re-assemble and re-compile using the makefile procedures provided with the C Library source. To use the new object files, either explicitly link your program with the new objects, or replace the Crt0dat.Obj (and possibly _file.Obj) in the appropriate model of your C Library and replace the ApiFile.Obj in LIBCEXT.LIB.

Section 3
Using the Programmer's WorkBench

This section introduces the Programmer's WorkBench (PWB), an integrated tool designed to make writing and debugging programs easier than using separate command line utilities. PWB is a window-oriented programming environment that incorporates a text editor, compiler, linker, debugger, Make utility, source-code browser, and online Help.

Since PWB is very large and has many features, this section does not document all features. It does, however, describe how to use menus and commands, step you through the process of creating .Run files, and introduce the Browser and CodeView, two utilities used to debug programs. To demonstrate these features, this section includes a sample program for you to compile, link, and debug called IO.c, located in the subsection Compiling and Linking.

This section covers the following topics:

- Starting PWB
- Using Windows and Menus
- Using the Editor
- Compiling, Linking, and Running Programs
- Debugging Programs

For more information on how to customize PWB, refer to Section 12, Customizing Programmer's WorkBench. Section 12 contains information on how to change Editor settings and keystrokes, how to write and use macros, and how to write PWB Extensions in C. For complete reference information on PWB features, refer to online Help.

Note: If you are familiar with the DOS version of Microsoft C, note that for CTOS the ALT key maps to COPY, and CRTL maps to CODE. In addition, pseudofiles for the DOS version are enclosed with angle brackets < >. For CTOS the pseudofiles are enclosed in parentheses ( ). For example, <assign> maps to (assign).

Starting PWB

Before starting PWB, you must first install Task Management Service. PWB and Install Task Management Service are both Executive commands.

To start PWB, follow this procedure:

1. Install Task Management Service using one of these methods:
   - Execute the Executive command **Install Task Managment Service**.
   - Add this line to SysInit.Jcl and reboot:

     $RUN [*disk*]<*directory*>TMS.RUN
2. Type **PWB** on the Executive command line, and press **GO**.

   The initial PWB screen appears.

Note: If you installed Environment Service, you should disable CL and MLINK environment variables. These variables can cause unpredictable build behavior under PWB, because PWB reads TOOLS.INI and CURRENT.STS to get environment variables, which may be different thant those set with Environment Service.

You can now write source code in the new, untitled window. In addition, PWB maintains a history of the files that you open and save, so the next time you start PWB, the last file you worked on appears.

With PWB Executive command options, you can also specify files to appear upon startup other than the one last edited.

To open a source file other than the one last edited, follow this procedure:

1. Type **PWB** on the Executive command line, and press **RETURN**.

   The PWB command form appears.

```
PWB
        [Args]
        [     ]
```

2. In `[Args]` field, type the name of the file to open, and press GO.

   PWB displays the file for editing.

If PWB can not find the specified file in the current directory, it prompts you to create a new file. You can also open any file in PWB by selecting **File, Open**.

PWB also supports many startup command-line options. For details, refer to Appendix B, Command Reference.

Using Windows and Menus

This subsection introduces the components of the PWB environment, describes how to use Help, and covers these topics:

- Windows
- Menus
- Dialog Boxes
- Getting Help

Note: *You can enter commands in PWB with the keyboard or a mouse. Unless the right mouse button is mentioned, 'click' means to press and release the left mouse button once.*

Windows

Figure 3-1 shows the elements of a typical PWB screen. Some elements provide information only. For example, if CAPS LOCK is on, the letter C appears in the lower right corner of the screen in the Status Indicators area; the letter C is informational only. Other elements perform actions triggered by a specific keystroke or mouse action. For example, if you click on the Close button in the upper left corner of a window, the window closes. The button is active rather than informational.

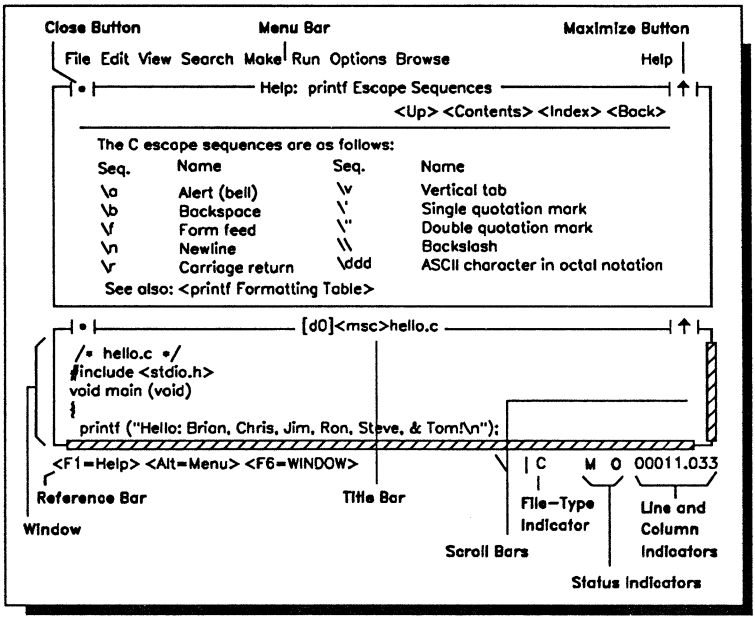


Figure 3-1. Parts of a PWB Screen.

Table 3-1 describes the screen elements shown in Figure 3-1.

Table 3-1. Parts of a PWB Screen

Name	Description
Menu bar	Lists names of available menus.
Close button	Closes window (appears only if more than one window is open).
Title bar	Shows the name of the file being edited.
Window	Contains source code or displays information associated with online Help.
Maximize button	Enlarges or restores window to its original size.
Scroll bars	Indicate cursor position in the current file and allow cursor movement.
Reference bar	Lists shortcut keystrokes (keyboard users) and directs PWB commands (mouse users); summarizes menu contents and displays other information.
File-type indicator	Indicates type of file: C: C source. text: any other user-created file. pseudo: file-like way to display data.

continued

Table 3-1. Parts of a PWB Screen (cont.)

Name	Description
Line and column indicators	Shows current line and column of text cursor.
Status indicators	A: Meta condition is on. C: CAPS LOCK is on. L: Carriage return is not used to terminate lines. M: File has been modified. N: NUM LOCK is on. O: Overtype is on. R: The file is set to Read Only status. T: File is temporary. X: A macro is recording.

Menus

The menu bar contains up to nine menus that you can use at any time except the Browse menu, which requires special Browse files created at compile time. In addition, the Browse menu may not appear if the Browse extension (pwbrowse.cxt) is not loaded in the same directory as PWB.Run.

This subsection covers the following topics:

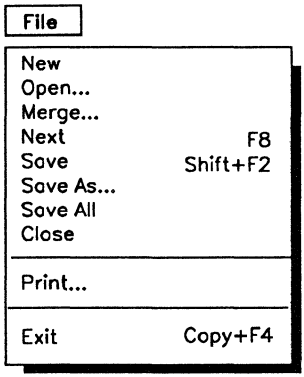
- Overview of Menu Commands
- Choosing Menu Commands
- Closing Menus
- Using Shortcut Keys
- Shaded Commands

Overview of Menu Commands

For reference, Table 3-2 provides a functional overview of menus and associated commands. For additional details about a menu or command, invoke online Help. To display Help about a menu item, press COPY, press the first letter of a menu item (appears bold), then press F1. For a brief description of a selected command, refer to the reference bar at the bottom of the screen. With the mouse, click Right on the menu, command, or reference bar.

Note: *If you are familiar with the DOS version of Microsoft C, note that for CTOS the ALT key maps to COPY, and CRTL maps to CODE.*

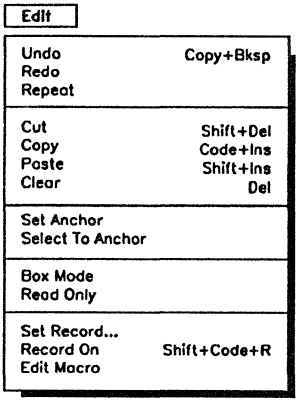
Table 3-2. Command Overview

Menu	Functional Overview
	The File menu allows you to: • Clear the Source window. • Load an existing source file. • Merge a file with the file in memory. • Display the next file in the file list specified at startup. • Save the current file. • Save the current file with a different name. • Print the selection or current file contents. • Exit PWB. In addition, a list of recently-opened source files appears at the bottom of the menu beneath Exit.

continued

Table 3-2. Command Overview (cont.)

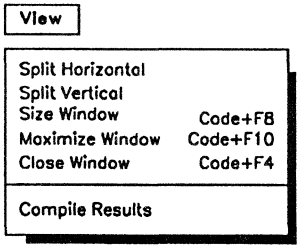
Menu	Functional Overview



The Edit menu allows you to:

- Delete, move, and copy text.
- Set anchors in the text for reference points.
- Block text using box, stream, or line mode.
- Use Read Only status to protect source files.
- Record, play, and edit recorded macros.

For more information on using Editor commands, refer to the subsection Using the Editor.

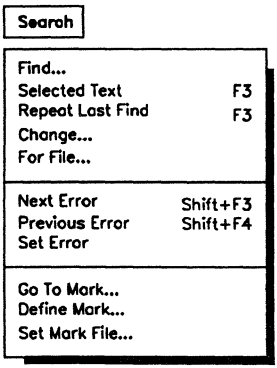
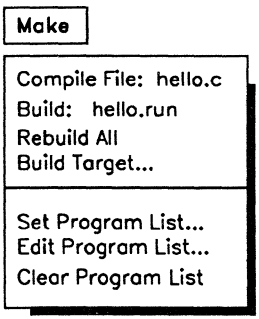


With the View menu you can:

- Split windows horizontally and vertically.
- Size and close windows.
- View compiler results.

continued

Table 3-2. Command Overview (cont.)

Menu	Functional Overview
	The Search menu invokes commands that: • Find or change text in source files. • Search source file for compile errors. • Use bookmarks to identify commonly-accessed text. Once you place a bookmark in source files, the Go To Mark command moves the cursor to that location.
	Use the Make menu to: • Compile and build programs. • Define PWB makefiles with Set and Edit Program List. • Load non-PWB make files, such as NMAKE files.

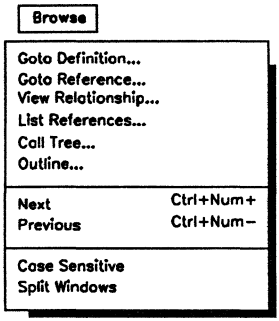
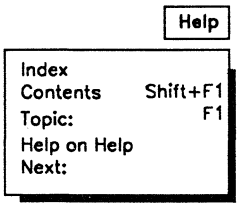
continued

Table 3-2. Command Overview (cont.)

Menu	Functional Overview
Run Execute: hello.run Command Line... Debug: Run Application Customize Menu...	Use the Run menu to: • Execute compiled programs. • Enter command line parameters if your programs take them. • Debug compiled programs with CodeView using the Debug command. • Run applications from PWB. • Add commands to execute other applications from PWB.
Options Environment... Key Assignments... Editor Settings... Build Options... LINK Options... NMAKE Options... CodeView Options... Browse Options... C Compiler Options...	From the Options menu, you can: • Define search paths for include, library, and Help files. • Customize key assignments and Editor settings. • Specify the build directory and these options: debug and release build options, Run or Extension executable files. • Set options for the C Compiler, Linker, Browser, NMAKE, and CodeView.

continued

Table 3-2. Command Overview (cont.)

Menu	Functional Overview
	The Browse menu controls the Browser debugger, which is activated after you: • Compile in PWB with Browse Options set to [X] Generate Browse Information. • Command-line compile with CL using switch /FR, which creates the file *source*.SBR; then run the command PWBRMake and specify *source*.SBR as an argument, which generates the other required file *source*.BSC. To use the Browser, first verify that the *source*.c, *source*.SBR, and *source*.BSC files are in the same directory; then open *source*.c in the Editor. Also, a makefile must exist for *source*.c and be selected from **Options**, **Set Program List**. The Browse menu is then activated
	The Help menu displays online Help. For more information, refer to Section 4, Using Online Help.

Choosing Menu Commands

To use menu commands, you first open (pull down) the menu from the menu bar, then choose commands.

To choose commands with keystrokes:

1. Press **COPY** to activate the menu bar.

   The first letters of each menu name appear highlighted.

2. Press the highlighted character in the menu name (such as **F** for File).

   The menu descends.

3. Press Up and Down Arrow keys to highlight the desired command, and press **GO**.

   Either the command executes, or a dialog box appears if ellipsis (. . .) follow the command.

To choose a command with the mouse:

1. Click Left on the menu name.

   The menu descends.

2. Click Left on the command.

   Either the command executes, or a dialog box appears if ellipsis (. . .) follow the command.

Closing Menus

If you open a menu and then decide not to issue a command, you can close the menu two ways:

- Press CANCEL.
- Click the mouse somewhere outside of the menu.

Using Shortcut Keys

Certain menu commands are followed by the names of keys or key combinations called shortcut keys. Shortcut keys exist for commonly-used commands, and without opening menus, allow you to execute commands with one keystroke. For example, the Search menu contains the Repeat Last Find command. To the right of the command appears F3. To issue this command at any time, press F3.

Note: *The reference bar at the bottom of the screen also displays common shortcut keys. To execute commands from the reference bar, click Left on <command>.*

Shaded Commands

When a command in a menu appears gray, it is unavailable. Some conditions prevent you from using commands. For example, when you start PWB for the first time, before you compile or link, the Next Error and Previous Error commands in the Search menu are unavailable.

Dialog Boxes

When selecting commands followed by ellipses (...), a dialog box appears; otherwise, commands execute immediately. Dialog boxes prompt you for more information that the command uses. For example, the Search menu's Find command opens the dialog box shown in Figure 3-2.

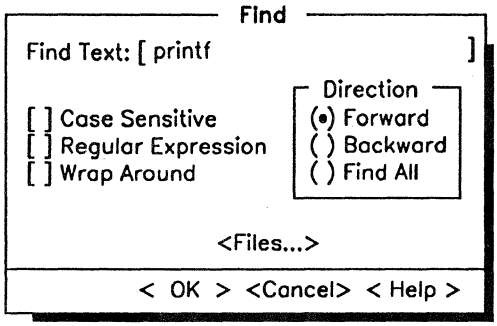


Figure 3-2. Dialog Box for the Find Command

Note: Dialog boxes usually offer shortcut keys. Shortcut keys appear as highlighted letters, and can be activated by pressing the corresponding key. For example, to select Case Sensitive in the Find dialog box, press C, the highlighted letter.

Dialog boxes can contain one or more of the items listed in Table 3-3. To move from item to item with keystrokes, either press the shortcut keys, TAB, or SHIFT-TAB. The fastest way to select items, however, is to click the mouse on the items. Table 3-3 describes keystrokes that manipulate dialog box features; it does not describe clicking the mouse.

Table 3-3. PWB Dialog Box Features

Item	Description
Option buttons (•)	Identifies an option in a list where you can choose one option only. To move from option to option with keystrokes, press the option shortcut keys, then press Arrow keys to move the dot. For example, Figure 3-2 shows that you can choose one search direction: Forward, Backward, or Find All.

continued

Table 3-3. PWB Dialog Box Features (cont.)

Item	Description
Check box [X]	A yes/no switch. If the box is empty, the option is off. If X appears, the feature is on. To toggle a check box on and off, either press the Spacebar; or press Up or Left Arrow keys to toggle on, Down or Right Arrow keys to toggle off.
Text box [. . . . .]	Accepts text that you enter. For example, in Figure 3-2, Find Text requires you to enter text to search for.
Command buttons < OK > < Cancel >	Pass commands to the dialog box (enclosed in angle brackets called buttons). < OK > acknowledges the current settings. <Cancel> exits the dialog box and does not change current settings. To execute a command from a highlighted button, press GO. If a button contains an ellipsis, a dialog box appears first.
List boxes	Display lists of information inside a box, such as the contents of the current disk directory. To display items not appearing in the allocated space, press Up or Down Arrow keys, NEXT PAGE or PREV PAGE, or click on the scroll bar.

Getting Help

PWB uses the CTOS Microsoft Advisor Help system to provide information about PWB, the compiler, the Linker, the CodeView debugger, other utilities, and topics relating to the C language. You can display information at any time from PWB or CodeView.

This subsection summarizes how to display:

- Menu Help
- Dialog Box Help
- The Help Menu

For complete details, refer to Section 4, Using Online Help.

Menu Help

Menus and commands have context-sensitive help.

To display Help for menus and commands, highlight a menu or command and either:

- Press **F1**.
- Click the right mouse button.

A Help window appears.

Dialog Box Help

Many dialog boxes have a Help button that provides additional information about the dialog box and its contents.

To display dialog box Help, either:

- Press **F1**.
- Click on < Help >.

The Help Menu

To display an index of all Help topics, a table of contents, and information about using the online reference system itself, use the commands listed in the Help menu.

To display a global contents screen for the entire Help system, select the Contents command. Contents lists information by topic, such as PWB, CodeView, and the C language.

Using the Editor

The program Editor is an important part of the PWB environment. This ubsection provides a brief overview of its major functions, and assumes you are using default key assignments. For a complete description of functions and features, refer to online Help.

This subsection covers the following topics:

- Moving around in a Source File
- Defining a Block
- Setting Bookmarks
- Setting Anchors
- Searching for and Changing Text
- Creating Macros

Moving around in a Source File

To move around source files, you can use either keystrokes or a mouse. To move around with keystrokes, press the standard keys:

- PREV PAGE
- NEXT PAGE
- SCROLL UP
- SCROLL DOWN
- Arrow keys
- CODE+E
- CODE+B

With a mouse you can move around in source files by clicking on scroll bars that appear at the bottom and right side of the editing window.

To move one row or column at a time:

- Click on arrows at the end of the scroll bars.

To move one window at time:

- Click on either side of the box that appears inside the scroll bars.

To precisely control how far to move in the file:

- Click, hold, and drag the box that appears inside the scroll bars.

Defining a Block

Blocks are highlighted areas of text that you select. A block can be as small as one character, or as large as the whole file. Generally, however, you will select blocks of a few words to a few lines.

Once you select a block, you can delete, move, and copy it using commands from the Edit menu.

> **Caution:**
>
> Once you highlight a block of text, if you press any key other than ones to pull down menus or execute commands, the block disappears and is replaced with the typed character.

To define a block with keystrokes:

1. Move the cursor to the beginning of the block and press MARK.
2. Move the cursor to the end of the block and press BOUND.

   To deselect the block, press CODE-MARK.

To define a block with the mouse:

1. Move the cursor to the beginning of the block.
2. Press and hold the left button, drag and release.

PWB supports three types of blocks: Box, Line, and Stream. The type that appears depends on how X Mode appears, a command located in the Edit menu where X is either Box, Line, or Stream.

Box mode	Selects rectangular blocks.
Line mode	Selects rows of text.
Stream mode	Selects all text.

The mode that appears in the menu is the next available mode.

You can also toggle through the selection modes when blocking text by holding down the left button and clicking Right. If you block text and use this method, you will quickly discover how to distinguish between the three modes.

Setting Bookmarks

A bookmark is a name to jump to that you can place in source code using Search, Go To Mark. You can set bookmarks anywhere in your source code with Search, Define Mark. You can also set and jump to bookmarks in different files other than the current file.

To set a bookmark, follow this procedure:

1. Move the cursor to the location you want to mark, and select **Search, Define Mark.**

   A dialog box appears, and displays the source file name and cursor position, as shown in Figure 3-3.

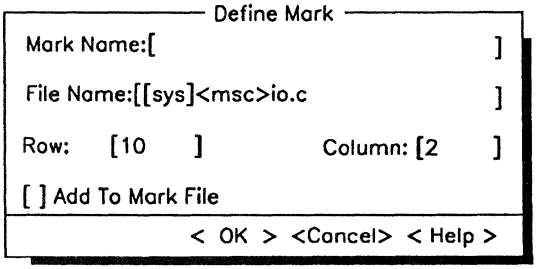


Figure 3-3. Define Mark Dialog Box

2. Enter a bookmark name, and select **< OK >.**

   The dialog box disappears.

3. To permanently set a bookmark, select **[x] Add to Mark File,** before selecting < OK >.

   Note: *This option works only if you have selected* ***Search, Set Mark File,*** *and entered a file name to store the settings. Otherwise, the bookmarks disappear when you exit PWB.*

To move to a bookmark, follow this procedure:

1. Select **Search, Go To Mark**

   A list of all bookmarks appears.

2. Select a bookmark from the list using the Arrow keys, or click on it with the left mouse button.

3. To move to the bookmark, select **< OK >.**

   The bookmarked place in source code appears, even if in a different file.

Setting Anchors

An anchor is a temporary marker that identifies the beginning of a text block. It offers a convenient way to start a block of text when you are not sure where the block will end.

To set an anchor, follow this procedure:

1. Move the cursor to the start of the block, and select **Edit, Set Anchor.**

2. Move the cursor up or down from the anchor, and select **Edit, Select to Anchor.**

   The text between the anchor and the current cursor position becomes blocked.

You can manipulate this block of text like any other block: move, copy, or delete. You can set one anchor at a time only.

Searching for and Changing Text

PWB allows you to search for and replace strings of text.

To search for a string of text, follow this procedure:

1. Select **Search, Find.**

   The Find dialog box appears.

2. Enter the string to look for, and select any other options that refine or direct the search pattern.

3. To search multiple files, select **< Files . . . >.**

   The Search in Selected Files dialog box appears, listing all files in the current directory.

   - Find and highlight the files to search, then select **< Add/Delete >.**

     The files appear in the File(s) Selected box.

4. Select **< OK >.**

   The number of occurrences appears in the lower left corner of the screen.

5. To move to the first occurrence in the current file only, select **Search, Repeat Last Find**, or press **F3.**

   The first occurrence of the string appears.

6. To move to the next occurrence of a string in the current file only, select **Search, Repeat Last Find** or press **F3.**

7. To move to the next occurrence of the string in a different file (if you specified several files to search), select **Search, Next Error** or press **SHIFT-F3.**

To replace a string , follow this procedure:

1. Select **Search, Change**.

   The Change dialog box appears.

2. Enter the target string, the replacement string, and search options.

3. To replace the string in the current file only, select either **< Find and Verify >** or **< Change All >**.

   If you select Find and Verify, the system prompts you to respond Yes or No before changing each occurrence. If you selected Change All, the system changes all occurrences without prompting you.

   Note: *Steps 4 through 7 apply to replacing strings in multiple files only.*

4. To replace the string in multiple files (assuming you skipped step 3), select **< Files . . . >**.

   Another Change dialog box appears, listing all files in the current directory.

5. Find and highlight the files to replace strings in, then select **< Add/Delete >**.

   The files appear in the File(s) Selected box.

6. Select **< OK >**.

   The first occurrence of the string appears, along with the Search Change dialog box.

7. Choose one these options:

   - To replace the string, select **< Yes >**.

     The change occurs and the next string appears.

   - To skip this replacement and move to the next string, select **< No >**.

     The next string appears.

   - To replace this and all other occurrences, select **< Change All >**.

     All strings are replaced in all files.

PWB also supports search-and-replace of text patterns using regular expressions. Both UNIX and non-UNIX regular expression syntaxes are supported. For further information, refer to online Help.

Creating Macros

A macro is a sequence of keystrokes that you can record and then replay at any time by pressing one key or a combination of keys. For example, when tracking source code changes by programmer, you may find that adding descriptive comments takes time, especially if your change template requires a large amount of information.

To eliminate having to type the generic comment information, you could create a macro that, with one or two keystrokes, creates a comment template and automatically enters the date and programmer's name. Thus, all the programmer needs to do is run the macro and type a description of the change.

Creating and using a macro involves three procedures: defining, recording, and replaying.

To define a macro, follow this procedure:

1. Select **Edit, Set Record.**

   The Set Macro Record dialog box appears.

2. Complete the dialog box:

   - In the Name: field, type a descriptive name for your macro.
   - In the Key Assignment field, position the cursor inside the curly braces { } and press the keys to execute the macro.

     The keystrokes appear between the square braces [ ].

   - To to append keystrokes to an existing macro, deselect the checkbox [ ] Clear First.

3.. Select **< OK >.**

   The menu disappears

To record a macro, follow this procedure:

1. Select **Edit, Record On.**

   PWB is now recording keystrokes.

   Note: *If you pull down the Edit menu, you will notice that a bullet now appears left of Record On. This indicates that PWB is recording keystrokes. An X also appears on the status line. The bullet and X disappear when PWB stops recording keystrokes.*

2. Press the keystrokes to record.
3. To stop recording, select **Edit, Record On** again.

   PWB stops recording keystrokes.

To execute the recorded keystrokes:

- Press the keystrokes that you assigned when defining the macro.

For full details on how to use macros, refer to online Help. (Highlight the macro commands and press F1 or click Right on it.).

Customizing the Editor

You can customize the PWB Editor to meet your needs. For example, you can modify Editor settings, and assign keyboard commands.

This subsection covers the following topics:

- Changing Editor Settings
- Modifying Keyboard Assignments
- Using Advanced Editor Features

In addition, you can write customized extensions to the Editor using the C language. For details about writing extensions, refer to Section 12, Customizing Programmer's WorkBench.

Changing Editor Settings

The PWB Editor has a variety of settings that you can customize, such as word wrap, color, and width of tab stops. PWB displays these settings from a pseudofile named (assign). A pseudofile is a temporary file. If you save (assign), PWB updates initialization files Tools.Ini and Current.Sts. For details about initialization files, refer to online Help, and to Section 12, Customizing Programmer's WorkBench.

The (assign) file lists numerous Editor settings and assigns values to them using the format `setting:value`, where *setting* is the name of the setting, and *value* is a Boolean, numeric, or text value. To learn about a setting using Help, position the cursor on the setting and press F1.

To view or modify Editor settings, follow this procedure:

1. Select **Options, Editor Settings.**

   A pseudofile named (assign) appears, titled Current Assignments and Switch Settings.

2. To change the value of a setting, replace *value* with a new value, then move the cursor to a different line.

   If the new value is legal, the changed line appears highlighted. If the new value is not legal, an error message appears.

3. To save the changed assignments, press **SHIFT-F2** (if the default keyboard assignments are in effect), or select **File, Save.**

   PWB updates the initialization files Tools.Ini and Current.Sts.

   Note: *If you do not save, the assignments are temporary and last for the current PWB session only.*

4. To return to your source file, press **F2** or select **File, Close.**

   Your source file reappears.

Modifying Keyboard Assignments

You can change any or all PWB keyboard assignments. PWB displays these settings from a pseudofile named (assign). This file now lists PWB functions and assigns keys to the functions using the format *function:keyname*. A list of unused keys follows the assignment list. To learn more about a PWB function, position the cursor on the function name and press F1.

To view or modify key functions, follow this procedure:

1. Select **Options, Key Assignment**.

   A pseudofile named (assign) appears. The title bar says Current Assignments and Switch Settings.

2. To assign a new key to a function, replace *keyname* with the name of the new key, then move the cursor to a different line.

   If the new key is legal, the changed line appears highlighted. If the new key is not legal, an error message appears.

3. To save the changed assignments, press **SHIFT-F2** (if the default keyboard assignments are in effect), or select **File, Save**.

   PWB updates the initialization files Tools.Ini and Current.Sts.

   Note: *If you do not save, the assignments are temporary and last for the current PWB session only.*

4. To return to your source file, press **F2** or select **File, Close**.

   Your source file reappears.

Using Advanced Editor Features

Most standard editing features used from menus and functions keys are intuitive and easy to use. However, you can set a variety of functions and switches to further customize the Editor and enhance its performance. The following procedure shows an example of an advanced Editor feature.

To assign a key command to the Editor's curdate function, which enters the current date at the cursor, follow this procedure:

1. Press **COPY-A**.

   An argument dialog box appears.

2. Enter **Curdate:SHIFT-CODE-T**.

3. To assign the key to the function, press **COPY-=**.

   The current date appears at the cursor position.

Now, whenever you press SHIFT-CODE-T, the current date appears at the cursor position. For a complete list of Editor functions and settings, refer to online Help.

Compiling, Linking, and Running Programs

PWB provides a complete build environment, and uses a project-oriented approach to building programs. You can compile, link, and run programs without leaving PWB.

This subsection covers the following topics:

- Procedural Overview
- Sample Program
- Saving Source Files
- Setting Compiler Options
- Setting Linker Options
- Saving Compiler and Linker Options as a Build Option
- Creating and Selecting Makefiles

- Selecting a Build Option
- Building Programs
- Running Programs
- Building Programs from Multiple Source Files
- Quick Reference

To follow the procedures contained in this subsection, we recommend that you create and use the sample program listed under the heading Sample Program. While most procedures contain generic steps, all illustrations are based on using this example. You will also find procedures to modify this sample file under the heading Building Programs from Multiple Source Files, and use the resulting file to follow a sample CodeView animation procedure found in the subsection Debugging Programs.

Notes: *For information on how to compile and link from the Executive command line, refer to Section 5, Building Applications.*

If you installed Environment Service, you should disable CL and MLINK environment variables. These variables can cause unpredictable build behavior under PWB, because PWB reads TOOLS.INI and CURRENT.STS to get environment variables, which may be different than those set with Environment Service.

Procedural Overview

Before building run files from PWB, you must choose compiler and linker options from six menus. From an initial menu for each utility, you choose general options, such as memory model. From two submenus displayed from each initial menu, you choose specific options separated into two categories: debug and release. Options used to create debuggable run files appear on one submenu; options used to create finished (releaseable) run files appear on the other submenu.

After you select options from all six menus, you save all options as one build option and name it. You can then recall your compiler and linker options at any time by selecting the build name. Once you select the build name, you can toggle between your debug and release options with a radio button.

Figure 3-4 shows how the menus relate to each other.

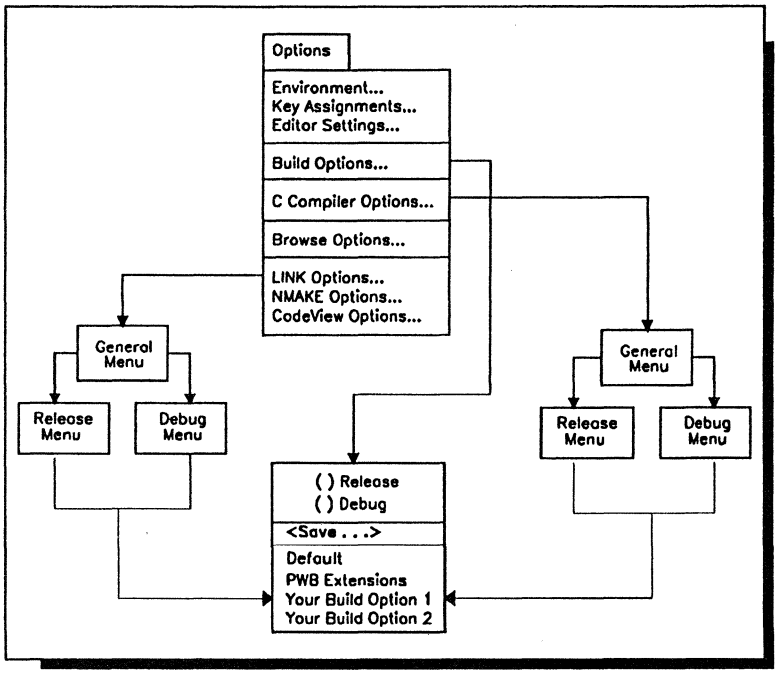


Figure 3-4. Setting and Saving Build Options

Before you can build run files, you must create a PWB-type makefile called a program list. To this file, you add the names of your source files as shown in Figure 3-5. You can also add any type of file to this list, including files required by the CodeView debugger such as source.cDV (which you must add to use CodeView from PWB).

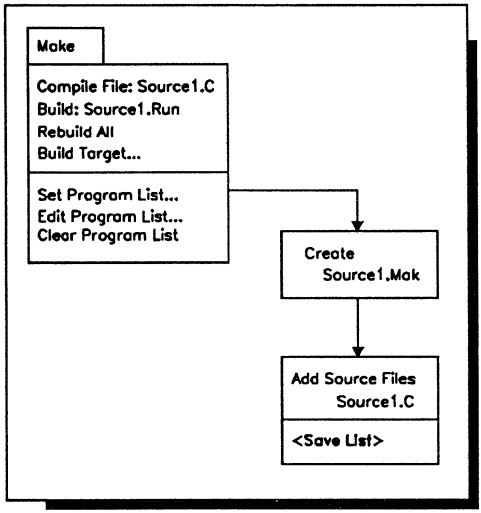


Figure 3-5. Creating a Makefile

Once you have completed this process, you can build a run file. The benefit of going through this process is that you can now build and rebuild with a three-step process as shown Figure 3-6: select your make file, set your build option, and build.

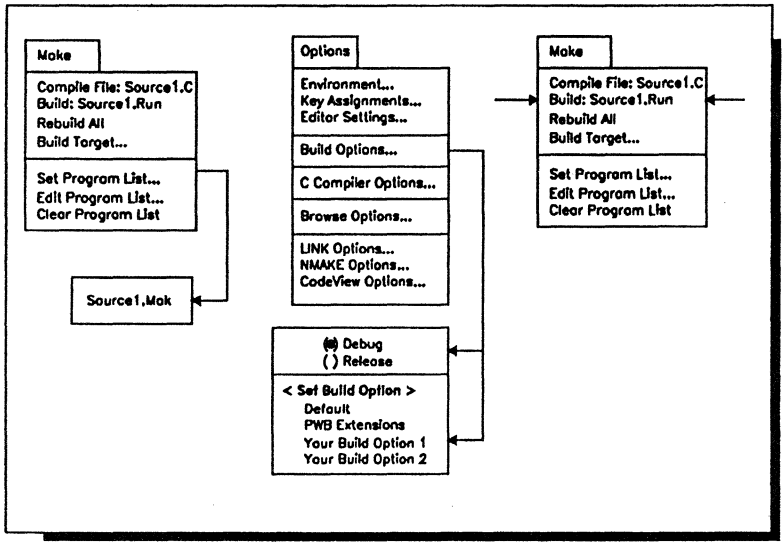


Figure 3-6. Building Programs

Sample Program

To follow along with all procedures listed in the rest of this section, type this program in an (UNTITLED) PWB screen (select **File, New**). Whenever you see *source*.c or *application*.c, you can substitute IO.c, the name of sample program. For example, if you see *source*.c in a procedure, substitute IO.c.

```
/* IO.c: String input and output */

#include <stdio.h>

main()

{

  char date[30];
  printf("Enter today's date: \n");
  gets(date);
  printf("Sample program run on %s\n", date);

}
```

This simple program uses the library functions `printf` and `gets` to prompt the user, read input, and display output to the screen.

Saving Source Files

To save source files, select File, and then select either Save or Save As. If you choose Save with an existing file, PWB automatically saves the file under the name listed in the title bar of the source window. To rename an existing file, choose Save As.

To save new source files, follow this procedure:

1. Select **File, Save As.**

   A dialog box prompts you for a file name as shown in Figure 3-7.

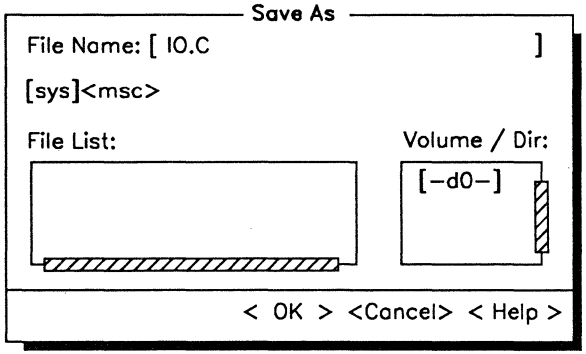


Figure 3-7. Dialog Box for the Save As Command

2. Enter the name of the source file, and select **< OK >.**

   PWB saves your file.

Setting Compiler Options

PWB provides menus that list many compiler options that you can select and deselect according to your needs.

To set compiler options, follow this procedure:

1. Select **Options, C Compiler Options.**

   The C Compiler Options menu appears, as shown in Figure 3-8.

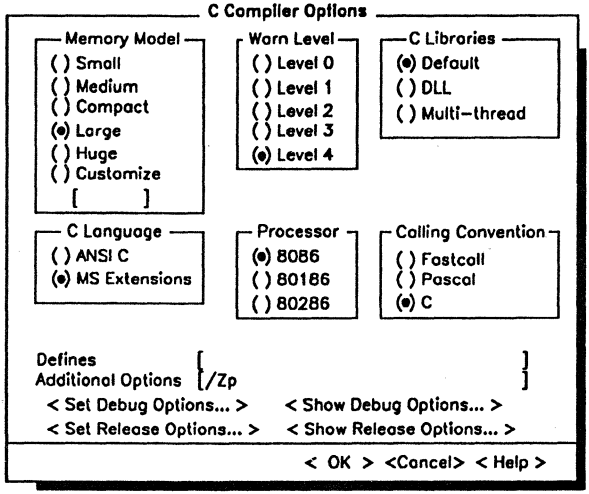


Figure 3-8. C Compiler Options Menu

2. Select the desired general compiler options, then select **<Set Debug Options>**.

   The C Compiler Debug Options menu appears, as shown in Figure 3-9.

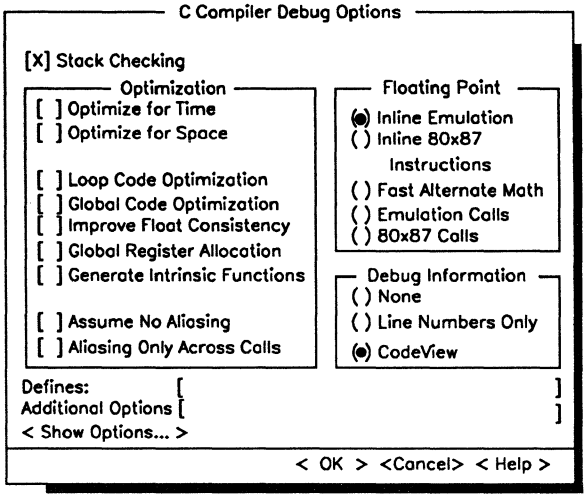

C Compiler Debug Options
[X] Stack Checking
Optimization
[ ] Optimize for Time
[ ] Optimize for Space
[ ] Loop Code Optimization
[ ] Global Code Optimization
[ ] Improve Float Consistency
[ ] Global Register Allocation
[ ] Generate Intrinsic Functions
[ ] Assume No Aliasing
[ ] Aliasing Only Across Calls
Floating Point
(•) Inline Emulation
( ) Inline 80x87 Instructions
( ) Fast Alternate Math
( ) Emulation Calls
( ) 80x87 Calls
Debug Information
( ) None
( ) Line Numbers Only
(•) CodeView
Defines: [ ]
Additional Options [ ]
< Show Options... >
< OK > <Cancel> < Help >

Figure 3-9. C Compiler Debug Options Menu

3. Select the options desired.

   Note: *Generally, you should not set optimizations for debug builds. If you are following along with sample program IO.c and want to use a procedure in the CodeView subsection that walks you through a sample animation session with IO.Run, set your debug options to match Figure 3-9.*

4. Select **< OK >** until the C Compiler Options menu reappears.

5. Choose **<Select Release Options>**.

   The C Compiler Release Options menu appears, as shown in Figure 3-10.

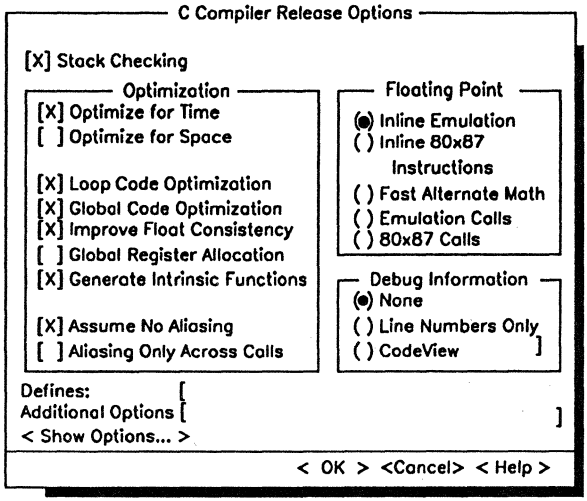


Figure 3-10. C Compiler Release Options Menu

7. Select the options desired.
8. Select **< OK >**.

   The C Compiler Options menu reappears.
9. When finished, select **< OK >**.

   PWB saves the options, and the editing screen reappears.

After you set these options, you can compile at any time by selecting **Make, Compile: *Source*.c**.

Setting Linker Options

PWB provides menus that list many linker options that you can select and deselect according to your needs.

To set linker options, follow this procedure:

1. Select **Options, LINK Options.**

   The Link Options menu appears, as shown in Figure 3-11.

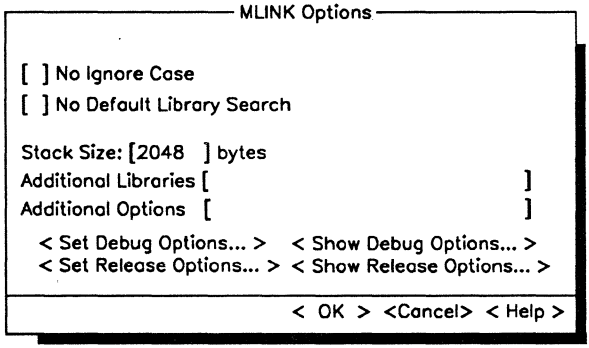


Figure 3-11. Link Options Menu

2. Select the desired general link options, then select **<Set Debug Options>**.

   The Link Debug Options menu appears, as shown in Figure 3-12.

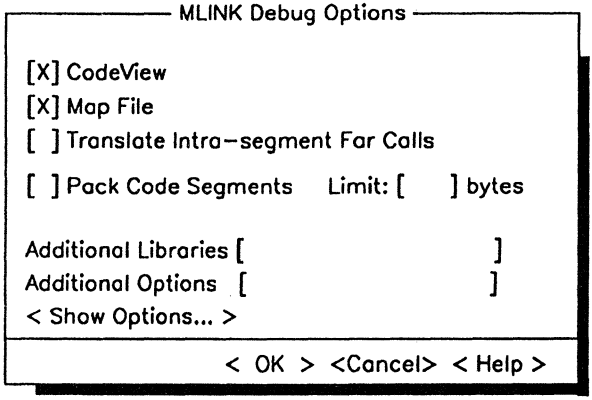


Figure 3-12. Link Debug Options Menu

3. Select the options desired.

   Note: If you are following along with sample program IO.c and want to use a procedure in the CodeView subsection that walks you through a sample CodeView session with IO.Run, set your debug options to match Figure 3-12.

4. Select **< OK >** until the Link Options menu reappears.

5. Select **<Select Release Options>**.

   The C Compiler Release Options menu appears, as shown in Figure 3-13.

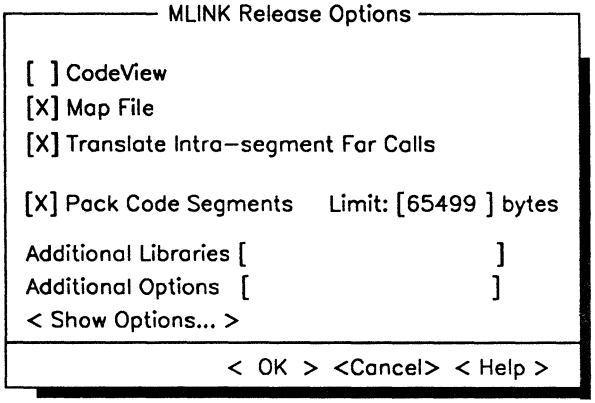


Figure 3-13. Link Release Options Menu

7. Select the options desired.
8. Select **< OK >**.

   The Link Options menu reappears.
9. When finished, select **< OK >**.

   PWB saves the options, and the editing screen reappears.

Saving Compiler and Linker Options as a Build Option

To save the compiler and linker options that you just set, you must name and save these options as one build option. This action allows you to recall all compiler and linker options at any time by choosing one build option name. Once you select a build option, you can swap between debug and release builds by clicking on radio buttons.

To save your compiler and linker options as a build option, follow this procedure:

1. Select **Options, Build Options.**

   The Build Options dialog box appears as shown in Figure 3-14.

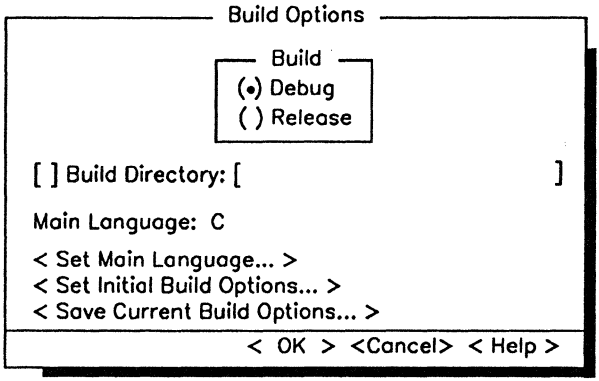


Figure 3-14. Build Options Dialog Box

2. Select **<Set Main Language>**, and select the C language.
3. Select **<Save Current Build Options>**, and enter a descriptive name for the build.

   Note: *To follow along with the IO.c example, enter* ***Testing.***
4. From the Build Options dialog box, select **< OK >.**

   PWB saves your compiler and linker options under the build option name you entered, and the Editor screen reappears.

Creating and Selecting Makefiles

Before PWB allows you to build or run a program you must create a makefile. PWB creates makefiles from Program Lists, which are simply lists of files. Once you create a makefile, you can select it at any time by selecting Make, Set Program List.

To create a makefile, follow this procedure:

1. Select **Make, Set Program List.**

   The Set Program List dialog box appears, as shown in Figure 3-15. All existing makefiles appear in the File List box.

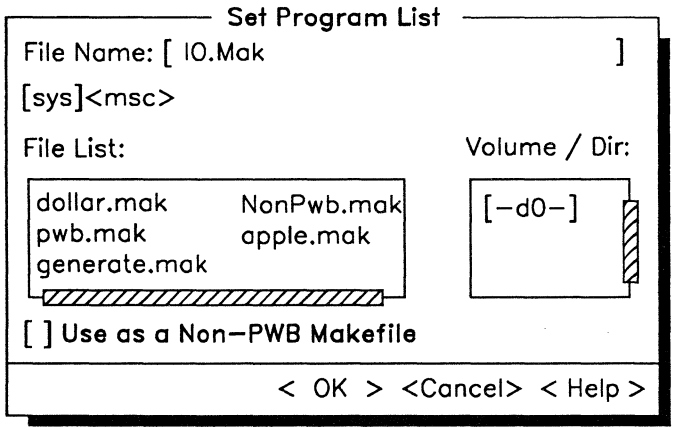


Figure 3-15. Dialog Box for the Set Program Command

2. Enter ***source*.Mak**, and select **< OK >**.

   A dialog box prompts you to create a new makefile, as shown in Figure 3-16.

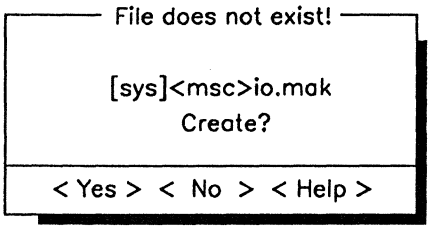


Figure 3-16. Makefile Dialog Box

3. Select **< Yes >**.

   The Edit Program List dialog box appears, as shown in Figure 3-17 listing all files in the current directory.

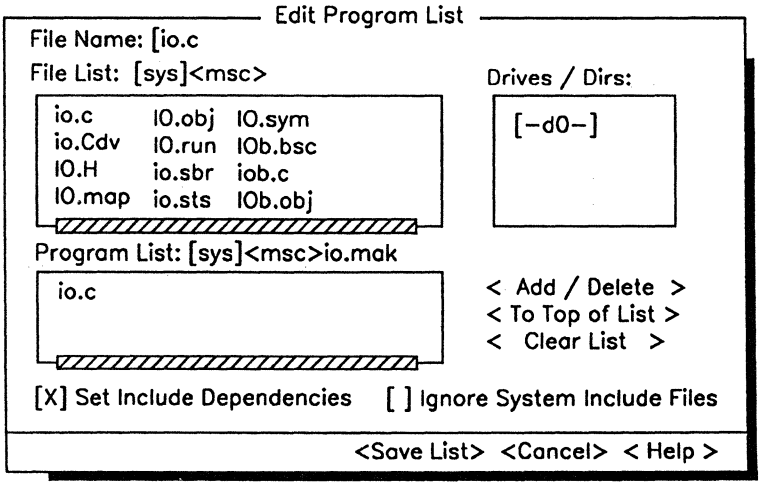


Figure 3-17. Edit Program List Command Form

4. Find and highlight *source*.c. (With the mouse, find the file and click on it. With keystrokes, press TAB and arrow keys.)
5. Select **< Add / Delete >**.

   Source.c appears in the Program List box.
6. Select **< Save List >**.

   PWB adds *source*.c to *source*.Mak, and the Editor screen reappears.

From now on, whenever you set options from the Options menu, PWB automatically updates the makefile when you Save.

Notes on Selecting Makefiles

As illustrated in the preceding procedure, all makefiles appear in the Set Program List dialog box. Whenever you compile or build files, be sure to select the appropriate makefile from this list if one exists, or else create one.

To use an NMAKE file if you recognize one in the File List box, first select **[X] Use as a Non-PWB Makefile**, then select the .Mak file. Be aware, however, that when selecting the Make menu for non-PWB makefiles, the file to build does not appear after the commands Compile and Build (from the Options pull-down menu) as it does when selecting PWB makefiles. Therefore, when using non-PWB makefiles you must select Rebuild All or Build Target.

Makefiles are especially useful with multi-module programs, because only changed source files are recompiled. To add multiple source files to a program list, use the Edit Program List command, and highlight and add the file. Using makefiles results in faster build times, since PWB does not recompile unchanged source files unless you select Rebuild All. For details, refer to Building Programs from Multiple Source Files, located at the end of this subsection.

Selecting a Build Option

The last step before buildling programs is to select a build option, which sets compiler and linker options to the values you chose earlier in this subsection. If you have been following the headings in this subsection, your build option name is already set. However, the concept of saving and selecting build options is important to understand, and if you are following along with IO.c, you need to select a debug version of your run file.

The build option menu initially appears with two build options: Default and CTOS PWB Extension. If you have been following the procedures with IO.c, you added a third option, Testing, in the procedure beneath the heading Saving Compiler and Linker Options as a Build Option.

To select a build option, follow this procedure:

1. Select **Options, Build Options.**

   The Build Options Dialog Box appears.

2. Select **<Set Initial Build Options>** ,

   The Set Initial Build Options dialog box appears, as shown in Figure 3-18.

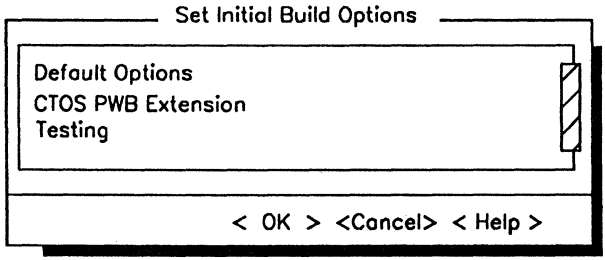


Figure 3-18. Set Initial Build Options Dialog box

3. Select an option, and select **<OK>**.

   The Build Options dialog box reappears.

   Note: *If you have been following along with IO.c, select* ***Testing****.*

   When building PWB extensions, select the Extension option, which sets the appropriate compiler and linker options. For details about extensions, refer to Section 12, Customizing Programmer's WorkBench.

4. Select either ( ) Debug or ( ) Release, and then select **<OK>**.

   Note: *To follow along with future CodeView procedures, select ( )* ***Debug****.*

5. To build object files and executables in a directory other than the current directory, select **[X] Build Directory**, and enter the path name.

   You are ready to build.

   Note: *If this option is disabled, specify a makefile from Make, Set Program List*

Building Programs

Once you set compiler and linker options, save them to a build option, and create a makefile, building a program is a one-step procedure. To watch PWB build your programs, select View, Compile Results before building, which opens a window at the bottom of the screen,

To build your source code into a Run file:

- Select **Make, Build: *application*.run**.

  PWB creates a Run file.

Note: *If application.run does not appear, select or create the appropriate makefile from Make, Set Program List.*

When compile or link errors occur, the build terminates, and the errors appear in the Compile Results window. For error explanations, highlight the error and either click Right or press F1. To jump to the place in code that caused the error, highlight the error, and click Left twice.

Running Programs

After you build programs, you can run them directly from PWB. You can also supply command line arguments to your programs from PWB. The following procedure assumes that you have selected the proper makefile, and have a successful build.

To set command line arguments and run a program, follow this procedure:

1. *Optional* To set command line arguments, select **Run, Command Line.**

   The Command Line dialog box appears.

2. *Optional* Enter your arguments, and select **< OK >.**

   The dialog box disappears.

3. Select **Run, Execute: *aplication*.Run.**

   PWB disappears, and your program executes.

4. To return to PWB, press any key.

   PWB reappears.

You can also run other applications by selecting **Run, Run Application.** However, this command is subject to limitations. You can run Executive commands and CTOS Microsoft C applications, but the command names can be one word only, and they can require no arguments.

Building Programs from Multiple Source Files

You can build applications from more than one source file. To use more than one source file, simply add the source files to the makefile from **Make, Edit Program List.** When you rebuild, PWB automatically incorporates the separate modules into the run file.

To show how to build programs from multiple source files, this subsection uses examples only, and covers these topics:

- Two Sample Source Files
- Rebuilding IO.Run with Two Modules

Two Sample Source Files

The following procedure demonstrates how to create a multi-module program, and is based on expanding the sample file IO.c, listed earlier in this section.

To create two sample source files, follow this procedure:

1. Edit the IO.c program to look like this:

```
       /* IO.c: Illustrates external functions */
#include <stdio.h>
void outsider(void);     /* Add this function.  */
void sample(void);       /* Add this function.  */

main()
{
   char date[30];

   sample();              /* Call new function.  */
   outsider();            /* Call new function.  */
   printf("Enter today's date: \n");
   gets(date);
   printf("Sample program run on %s\n", date);
}

               /* Implement one new function.  */
void sample(void)
{
   printf("A sample function.\n");
}
```

   The main( ) function now calls two new functions defined within the program: sample( ) and outsider( ).

2. Select **File, Save.**

3. Select **File, New**, and create this file:

```
/* IOB.c: Second module for IO.c */

#include <stdio.h>

void outsider(void);

void outsider(void)
{
printf("This line is from the file IOB.c.\n");
}
```

4. Select **File, Save As** and title it IOB.c.

Rebuilding IO.Run with Two Source Files

This subsection shows how to ad source files to a makefile using the example IO.c and IOB.c, and then shows how to rebuild IO.Run. In addition, the upcoming subsection Using CodeView contains a procedure that uses this rebuild to step you through an animation session.

To rebuild IO.Run from two source files, follow this procedure:

1. If you are not using IO.Mak, select it from **Make, Set Program List.**
2. Select **Make, Edit Program List.**

   The Edit Program List dialog box appears.
3. Find the file IOB.c in the **File List:** box, and highlight it.
4. Select **< Add / Delete >.**

   The file appears in the Program List: box.

5. Select **< Save List >**.

   PWB adds the source file to the makefile.

   Note: *This procedure assumes that IO.c already exists in IO.Mak. If IO.Mak is new, repeat steps 3 and 4 with IO.c=c before selecting < Save List >.*

6. Select **Make, Build: IO.Run**.

   PWB rebuilds the run file.

Note: *A program list can include source files (with the .c extension), assembler files (.ASM), object files (.Obj), and library files (.LIB).*

When you choose **Make, Rebuild All**, PWB compiles every .c file in the program list into an .Obj file. PWB then links all .Obj files with .LIB files to create one .Run file.

If you choose Make, Build, the NMAKE utility checks the time and date stamps on the source and object files. If the source code has not changed since the last time you executed a Build command, unchanged files are not recompiled. Consequently, the Build command is often faster than Rebuild All.

Quick Reference

The following procedural outline summarizes concepts described and keystrokes used to compile and link with PWB.

- Set Compiler Options

  Choose **Options, C Compiler Options**; set options.

  Choose **<Set Debug Options>**; set options; select **<OK>**.

  Choose **<Set Release Options>**; set options; select **<OK>**, **<OK>**.

  *Note: After you set compiler options, you can compile at any time by selecting **Make, Compile** **source.c**.*

- Set Linker Options

  Choose **Options, Link Options**; set options.

  Choose **<Set Debug Options>**; set options; select **<OK>**.

  Choose **<Set Release Options>**; set options; select **<OK>**, **<OK>**.

- Set and Save Build Options

  Choose **Options, Build Options**.

  Choose **<Set Main Language>**; select **C**; select **<OK>**.

  Choose **<Save Current Build Options>**; enter a ***build_name***.

  Select **<OK>**

- Create a PWB-type Makefile

  Choose **Make, Set Program List**; name the file ***source*.Mak**; select **<OK>**.

  Respond **<Yes>** to the new-file prompt.

  Move source files from the **File List:** to the **Program List:** windows.

  Select **<Save List>**.

- Select the Type of Run File to Build

  Choose **Options, Build Options**; select ( ) **Debug** or ( ) **Release**

  Choose **<Set Initial Build Options>**; select ***build_name***.

  Select **<OK>**, **<OK>**

- Build your Run File

  Select **Make, Build: *source*.run**

To build your run file at a later date:

Choose **Make, Set Program List**; select **source.Mak**; select **<OK>**.

Choose **Options, Build Options**.

Choose **<Set Initial Build Options>**; select ***build_name***; select **<OK>**.

Choose ( ) **Debug** or ( ) **Release**; select **<OK>**.

Choose **Make, Build: *source*.Run**.

Debugging Programs

PWB provides three utilities that you can use to inspect and debug programs:

- Error Messages and associated commands
- The Source Browser
- The CodeView debugger

When compiling or linking programs that cause errors, PWB lists and describes the errors, and provides commands that allow you to display the source lines that caused errors.

Before you can use the Browser or CodeView debugger, you must create special builds. To build for the Browser, you must create two files from your source code, which you can create from PWB, or from the Executive command line using two commands. To create build for CodeView, you must compile and link using CodeView options. You can set these switches and options from PWB or with Executive commands.

This subsection contains complete build information for the Browser, using both PWB and Executive commands. This subsection also describe how to create a debug build for CodeView using PWB. For details on how to set CodeView switches and options using Executive commands, refer to Section 13, Debugging with CodeView.

Working with Error Messages

When compile errors occur, the cursor moves to the first line that caused an error, and a brief description appears in the reference bar. PWB allows you work with these errors in several ways:

- To view a complete list of errors, select **View, Compile Results.**

  A window appears at the bottom of the screen, listing all warnings and errors. To view results as they occur when compiling, select this option before compiling.

- To synchronize error messages with associated source code lines, select **Search, Set Error.**

  As you highlight errors in the Compile Results window, the associated source code line appears in the Editor screen.

- To move from error to error, select **Search** and either **Next Error** or **Previous Error**. The associated shortcut keys are **Shift-F3** and **Shift-F4.**

  PWB highlights the next or previous error in both the Editor and Compile Results windows.

- To display online Help about an error from the Compile Results window, position the cursor on the error number, and either press **F1** or click Right.

Using the Browser

The Browser allows you to display information about data and function relationships within your source code, and locate references to all functions and variables in a multimodule program.

This subsection covers these topics:

- Preparing a Browser Build
- Inspecting with the Browser

Preparing a Browser Build

To use the Browser you must build two files, *source*.SBR and *source*.BSC, which contain Browser information. The Browse menu does not work until PWB detects these files.

You can create these files from both PWB and the Executive command line. To create the files from PWB, you set a menu option before compiling. To create the files from the Executive command line, you must compile with CL, and then run PWBRMake. The two procedures that follow describe both methods.

To create a Browser build from PWB, follow this procedure:

1. Open your source file.
2. Select **Options, Browse Options.**

   The Browse Options dialog box appears, as shown in Figure 3-19.

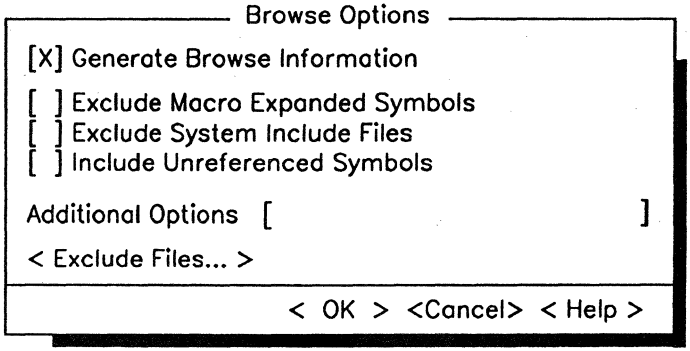


Figure 3-19. Browse Options Dialog Box

3. Select **Generate Browse Information**, and specify any other options.

   An X appears in selected check boxes.

4. Select **< OK >**.

   The Editor screen reappears.

5. To create the Browser files, select **Make, Compile**.

   PWB creates the files.

   Note: *This procedure assumes that you created a makefile with Set Program List, and set desired debug options with menu commands Build Options, C Compiler Options, and Link Options. For details, refer to Compiling and Linking, described earlier in this section.*

To create a Browser build using Executive commands, follow this procedure:

1. Compile your source files with the CL command using one of these switches:
   - /Fr Creates a standard Browser database
   - /FR Creates an extended Browser database

   CL creates the file *source*.SBR.

   Note: *The extended version contains all possible Browser information. The standard version contains truncated Browser information, but is slightly smaller only.*

2. Complete the PWBRMake command form as follows, and press **GO**.

   ```
   PWBRMake
         [Args]   source.SBR
   ```

   PWBRMake creates *source*.BSC.

   Note: *To save disk space, PWBRMake truncates .SBR files to zero length. To recreate a .BSC file, you must first recompile to create a new .SBR file, and then create a new .BSC file with PWBRMake.*

3. After you start PWB, add *source*.BSC to your makefile.

Inspecting with the Browser

Once you create Browser files, you can open your source files and use the Browse menu shown in Figure 3-20, assuming that:

- Browser files exist in the same directory as your source files.
- You created a makefile from **Make, Set Program List**.

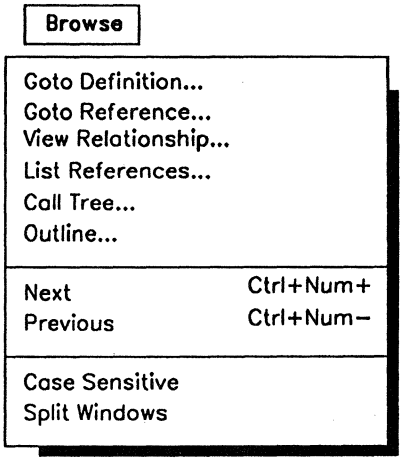


Figure 3-20. Browse Menu

Table 3-4 provides a brief overview of the top five commands in the Browse menu. For complete information on how to use the Browser, refer to online Help.

Table 3-4. Browse Commands

Command	Overview
Goto Definition	Provides a list of symbols and variables that occur within include files and modules. To open the file at the line number where the definition occurred, select **< OK >**.
Goto Reference	Displays a list of files and line numbers where symbols and variables have been referenced in include files and modules.
View Relationships	Provides detailed information about various portions of your program. You can examine functions, macros, types, and variables by their use, definition, or inclusion in include files and modules. The procedure that follows this table shows one way to use this command.
List References	Displays references to functions, macros, types, and variables. You can move the cursor to a reference, and then use the Goto Definition or Goto Reference commands to show occurrences of the reference.
Call Tree	Displays a tree structure of function calls within the program.

The following procedure shows how to display function calls made by main( ) in the sample program IO.Run, the bebuilt, multi-module version described under Building from Multiple Source Files. This procedure also assumes that you created a Browser debug build.

To display main( ) function calls for IO.Run, follow this procedure:

1. Open IO.c in the Editor.
2. Position the cursor on **main( )**.
3. Select **Browse, View Relationships.**

   The View Relationships menu appears and displays:

   ```
   . . .
   (•) Program Symbols
   . . .
   [X] Functions
   [X] Variables
   [X] Types
   [X] Macros.
   ```

4. In the **Show Only**: box, deselect:

   ```
   . . .
   [ ] Variables
   [ ] Types
   [ ] Macros.
   ```

5. In the **Operation:** box, select:

   ```
   . . .
   (•) Calls
   . . .
   ```

6. Select **< OK >**.

   All functions called by main( ) appear, as shown in Figure 3-21.

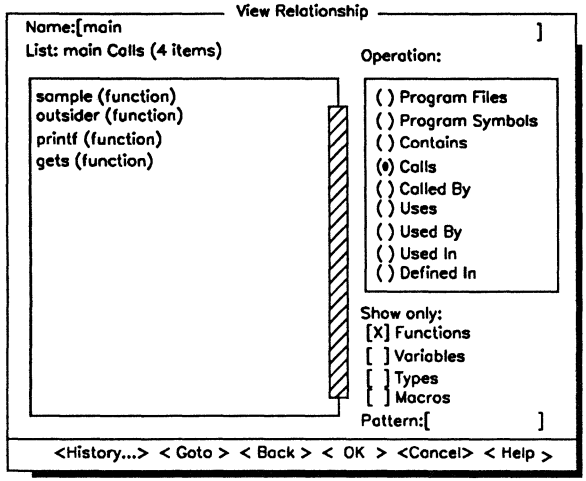


Figure 3-21. Browser Display of Functions

Using the CodeView Debugger

CodeView is a window-oriented symbolic debugger that lets you step through your code and examine variables and data as the program executes. With the CodeView debugger you can also set breakpoints, examine machine registers, and step through program execution line by line. This subsection provides an overview only, and describes how to compile for, link for, and move to CodeView from PWB.

You can also start CodeView from the Executive command line. For details about how to compile, link, and start CodeView using Executive commands, refer to Section 13, Debugging with CodeView. For complete details about all aspects of CodeView, refer to online Help.

Note: *For the CodeView debugger, the right mouse button does not display Help as it does in PWB. The right button sets breakpoints, unless you are inside Help where it works normally.*

This subsection covers these topics:

- Preparing a Debug Build
- Starting and Running CodeView
- Debugging IO.c

Preparing a Debug Build

To use CodeView you must build a run file and an associated *source*.Cdv file , which contains CodeView information. Even though CodeView works if it detects the .Run file only, you cannot view symbols or debug at the source level without the .Cdv file.

To create the .Cdv file, you must create a debug build. To create a debug build from PWB, you must:

- Set debug options for the compiler.
- Set debug options for the linker.
- Save your options to one build option, and select a debug build.
- Build a run file.

The procedures located in the previous subsection Compiling and Linking, describe how to configure and build debug versions of your programs for CodeView. In fact, the illustrations located under headings

Setting Build Options

Setting Compiler Options

Setting Link Options

show the menus and debug options to set for a successful CodeView build. Therefore, while the procedure that follows lists all steps necessary to build the CodeView file, it omits the details illustrated and described in the subsection Compiling and Linking.

Note: *If you are following along with IO.c, replace source with IO.*

Caution:

CodeView is not compatible with the CTOS Librarian because the Librarian modifies object modules during processing. Thus, modules compiled for CodeViewing and added to a CTOS library before linking cannot be CodeViewed at source level. MLIB does not modify object modules, so use MLIB to perform any and all library work.

To create a CodeView debug build from PWB, follow this procedure:

1. Select **File, Open** and open *source*.c.
2. Select **Options, C Compiler Options.**
3. Set compile debug options for CodeView, and select **< OK >.**
4. Select **Options, Link Options.**
5. Set link debug options for CodeView, and select **< OK >.**
6. Select **Options, Build Options**, and select **<Set Main Language>.**
7. Select **C.**
8. Select **<Save Current Build Options>**, name your build option, and select **<OK>.**
9. Select (•) **Debug**, and select **<OK>.**
10. Select **Make, Set Program List**, and create or select *source*.mak.
11. Select **Make, Build: *source*.run.**

    PWB builds both *source*.run and *source*.cdv.

Starting and Running CodeView

The procedure that follows describes how to start and exit the CodeView debugger, and illustrates a few CodeView features by stepping you through the IO.Run sample program in its multi-module form, described under the heading Building from Multiple Source Files.

Before starting CodeView from PWB, you may want to set startup options. PWB contains a menu that lists some startup options, and that provides a line where you can enter any startup option in the form of command switches. The menu is included in the next procedure.

Note: *To use the next procedure outside PWB and start CodeView from the Executive command line, use this command and start with step 3:*

```
CV
  [Args]   <Path>IO
```

To start CodeView and manipulate the sample file IO.Run, follow this procedure:

1. Select **Options, CodeView Options.**

   The CodeView Options menu appears, as shown in Figure 3-22.

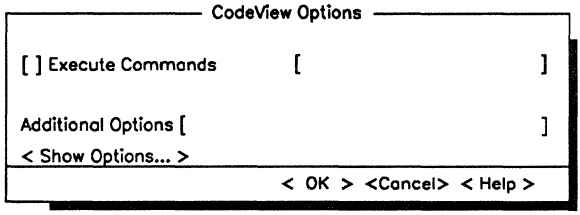


Figure 3-22. CodeView Options Menu

2. Select the desired options, then select **< OK >.**

   The Editor screen reappears.

3. To start CodeView , select **Run, Debug: *source*.run.**

   The initial CodeView screen appears, as shown in Figure 3-23.

Note: *If Assembly rather than C source code appears, press **F3** until C source appears.*

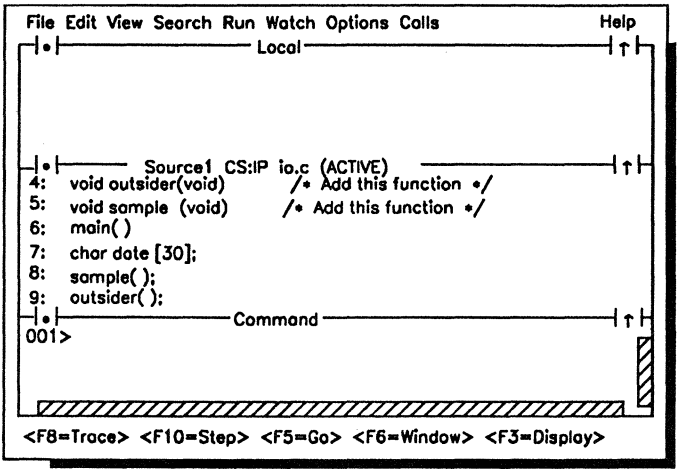


Figure 3-23. CodeView Startup Screen

4. To watch array variable date[30], place the cursor on **date[30]**, and select **Watch, Add Watch**.

   The Add Watch dialog box appears.

5. Select **< OK >**.

   The Watch window appears in the upper right corner, and displays date <Watch Expression Not in Context>.

6. Select **Options**.

   The Option menu appears.

7. If a bullet does not appear next to • Screen Swap, select **Screen Swap**. Otherwise, close the menu (CANCEL or click Left somewhere else).

   The menu disappears.

Caution:

This configures animation to swap to the debuggee after executing each line of code, so you can enter a date in step 9. If you do not set this option, your system will hang in step 9.

8. Select **View, Register**.

   The Register window appears on the righthand side of the screen.

9. Press **F10** twice (step).

   The date [ ] array appears in both the Local and Watch windows, and the highlighted bar stops on sample( ) as shown in Figure 3-24.

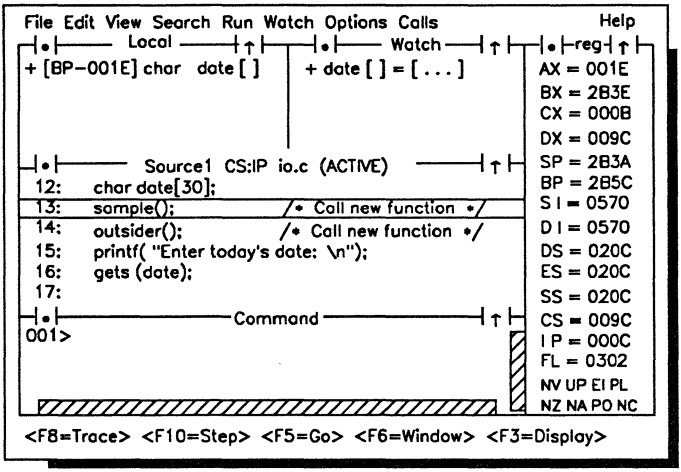


Figure 3-24. CodeView Screen with Watch and Register Windows

10. To expand the arrays in the Local and Watch windows, click Left on the plus (+) signs.

    The plus (+) signs change to minus signs (-), and the first few array elements descend, the elements that will receive the date.

    Note: *Anytime plus signs appear in front of arrays, you can expand the arrays. Anytime minus signs appear, you can contract the arrays.*

11. Press **F10** until the CodeView screen disappears, and following text appears on a blank screen:

```
A sample function.
This line is from the file IOB.c.
Enter today's date:
```

Note: If you did not set Screen Swap properly in step 5, your screen hangs at this point. Press ACTION-FINISH and start over.

12. For illustration purposes, type **2/23/92**, and press **RETURN**.

    The CodeView screen reappears and displays the array elements filled with the date values typed. CTOS character-set-equivalent values appear next to date values, as shown in Figure 3-25.

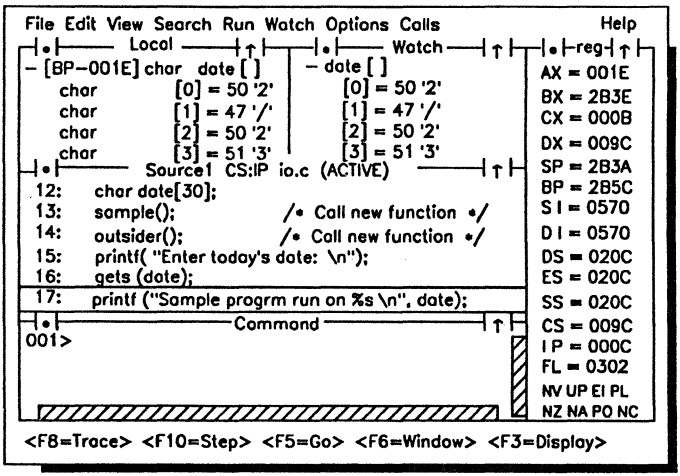


Figure 3-25. CodeView Screen with Expanded Arrays

13. Press **F10** until the program terminates.
14. Select **Files, Exit.**

    PWB reappears.

Debugging IO.c

To see how CodeView deals with a program bug, use the Exit command in the File menu to return to PWB, then add the following function to the IO.c program, and call it from main:

```
void bounds_error(void){
   int loop, the_num[10];
   for(loop = 0; loop < 15; loop++)
      the_num[loop] = loop; }
```

Set the appropriate debug options, rebuild the program, and return to CodeView.

Use the F8 Trace command to execute the program. The program runs until it reaches the `for` statement in the `bounds_error` function, where the array bounds are exceeded. The output window reports an error while a CodeView dialog box indicates the program has terminated. At this point you can return to PWB, locate the error, and correct it.

This Page is Blank

Do Not Print

Section 4
Using Online Help

CTOS Microsoft C provides an online Help system accessable from two utilities:

- QuickHelp, used from the Executive command line.
- The Advisor, used from the Programmer's WorkBench (PWB) and from the CTOS Microsoft CodeView Debugger.

Both utilities use the same database and provide the same information about PWB, the CTOS Microsoft CodeView Debugger, the CTOS Application Procedural Interface (API), and the C language. In addition, online Help shows syntax and examples for most C functions.

With few exceptions, both utilities look and function alike, and respond to mouse movements and keystrokes the same except when they display unmarked hyperlinks outside Help windows. Unmarked hyperlinks outside Help windows--function names, keywords, and so forth--exist in PWB source code and CodeView windows, and are explained in this section.

One major difference, however, lies in the pull-down menus. In QuickHelp, the pull-down menus are dedicated to the Help system. In PWB, the pull-down menus are dedicated to editing, compiling, and linking. In the CTOS Microsoft CodeView Debugger, the pull-down menus are dedicated to debugging.

Because the Advisor is basically a subset of QuickHelp without the pull-down menus, this section focuses on QuickHelp. In the list that follows, topics followed by an asterisk (*) identify subsections that apply to the Advisor.

This section covers the following topics:

- Running QuickHelp
- Structure of the Help System *
- Using Pull-Down Menus and Dialog Boxes
- Navigating through Help Screens *
- Copying and Pasting *
- Displaying Keyword Information *
- Displaying Error Code Information *
- Printing QuickHelp Information
- Configuring a Context Manager Partition

Running QuickHelp

This subsection covers the following topics:

- Starting QuickHelp
- Exiting QuickHelp

Starting QuickHelp

The QuickHelp Executive command takes the following form:

```
QH
   [Args] [options] topic
   [     ]
```

For details about options, refer to Appendix B.

To start QuickHelp, follow this procedure:

1. To direct QuickHelp to the directory of Help files (.Hlp), choose one of these options:
   - Configure EnvironmentConfig.Sys with **:HELPFILES:[*vol*]<*helpdir*>**, and execute the command **Install Environment Service** or use the **Environment Set** command after you install EVS.
   - **Path** to the directory that contains Help files before starting QuickHelp.
   - Specify the path on the QuickHelp command form with option **/d [*vol*]<*helpdir*>.**
2. On the Executive command line, type **QH**, and press **RETURN**.

   The command form appears.
3. Complete the command form if necessary, and press **GO**.

   The Help menu appears, as shown in Figure 4-1.

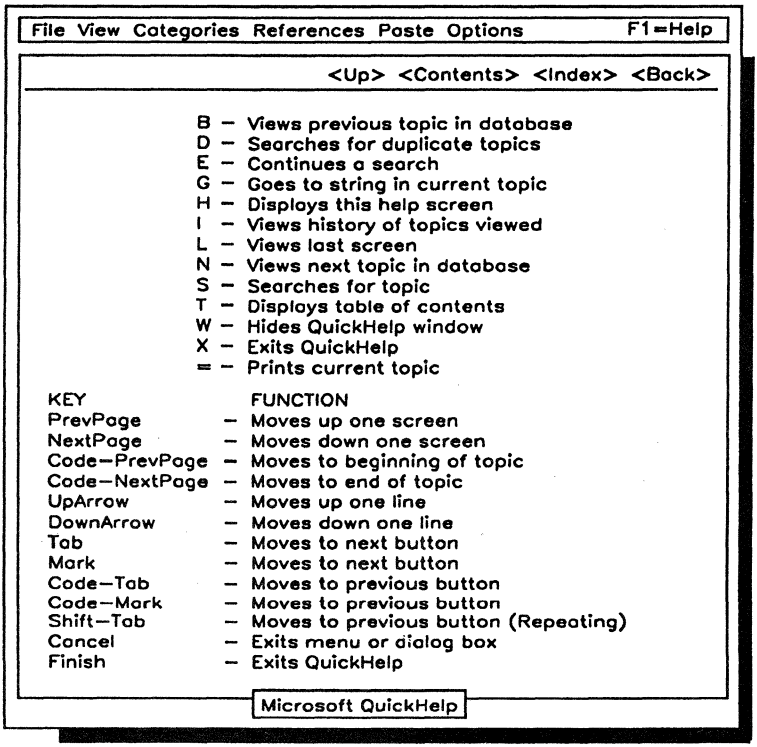

File View Categories References Paste Options F1=Help
<Up> <Contents> <Index> <Back>
B – Views previous topic in database
D – Searches for duplicate topics
E – Continues a search
G – Goes to string in current topic
H – Displays this help screen
I – Views history of topics viewed
L – Views last screen
N – Views next topic in database
S – Searches for topic
T – Displays table of contents
W – Hides QuickHelp window
X – Exits QuickHelp
= – Prints current topic
KEY FUNCTION
PrevPage – Moves up one screen
NextPage – Moves down one screen
Code–PrevPage – Moves to beginning of topic
Code–NextPage – Moves to end of topic
UpArrow – Moves up one line
DownArrow – Moves down one line
Tab – Moves to next button
Mark – Moves to next button
Code–Tab – Moves to previous button
Code–Mark – Moves to previous button
Shift–Tab – Moves to previous button (Repeating)
Cancel – Exits menu or dialog box
Finish – Exits QuickHelp
Microsoft QuickHelp

Figure 4-1. QuickHelp Initial Screen

Exiting QuickHelp

You can exit QuickHelp three ways.

To exit QuickHelp, choose one of these options:

- Click on **File, Exit.**
- Press **FINISH.**
- Press **X** (Exit

Structure of the Help System

The Help system can be compared to a librarian managing books. The system organizes all Help files with a global table of contents as shown in Figure 4-2. To display the contents screen from the initial QuickHelp screen shown in Figure 4-1, click twice on **<Contents>**. To display the contents screen from PWB or the CTOS Microsoft CodeView debugger, click on **Help, Contents**.

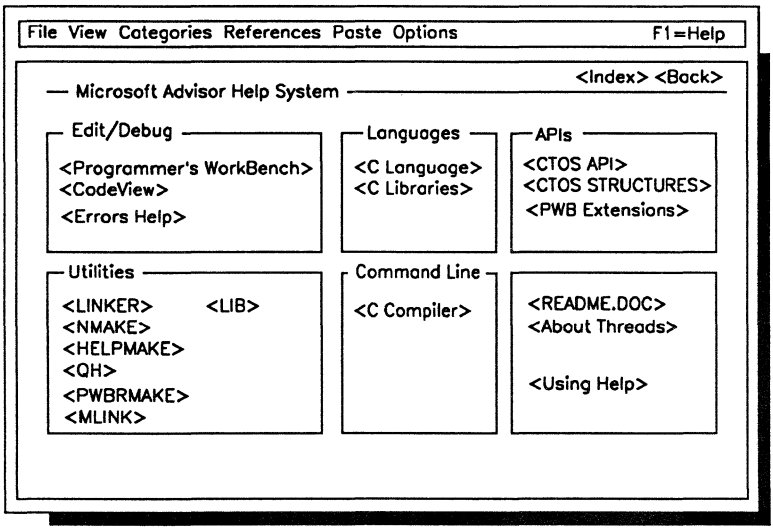


Figure 4-2. Global Contents Help Screen

The Help system also organizes Help files with a global index, as shown in Figure 4-3. To display the index screen from the screen shown in Figure 4-2, click on **<Index>**.

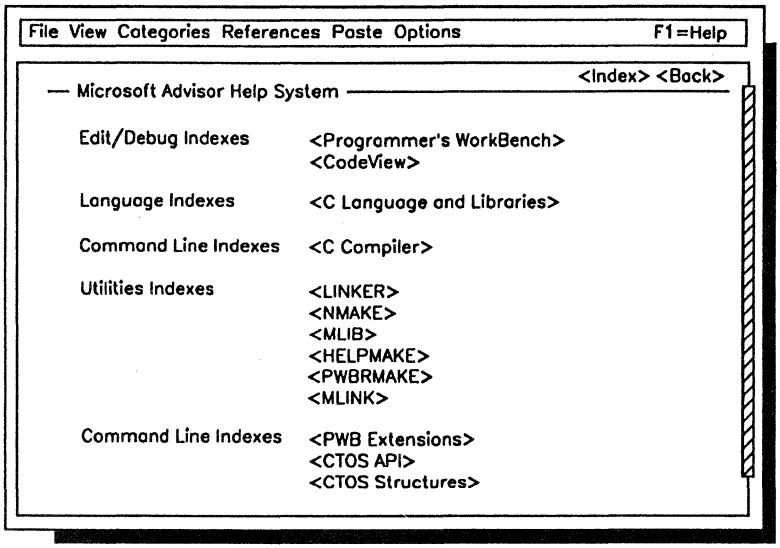


Figure 4-3. Global Index Help Screen

Each Help file also has a table of <Contents>, <Index>, and pages of information.

Using Pull-Down Menus and Dialog Boxes

The following subsections describe how to select commands from QuickHelp pull-down menus, and describe how to use dialog boxes that prompt for additional information when you use certain commands. To enter commands, use the mouse or keystrokes. When reading instructions to click the mouse, assume the left button unless the right button is mentioned.

Pull-Down Menus

The menu bar that appears at the top of all Help screens, contains six menus that you can pull down at any time.

To pull down a menu, either:

- Click on the title.
- Press the first letter of the title.

  For example, to open the File menu, press **F**.

The following figures shows the commands available from each menu.

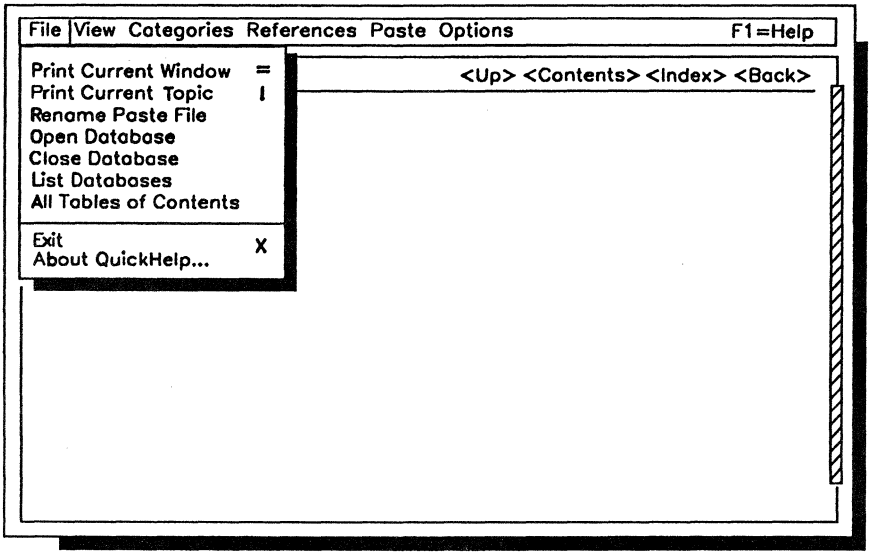


Figure 4-4. File Commands

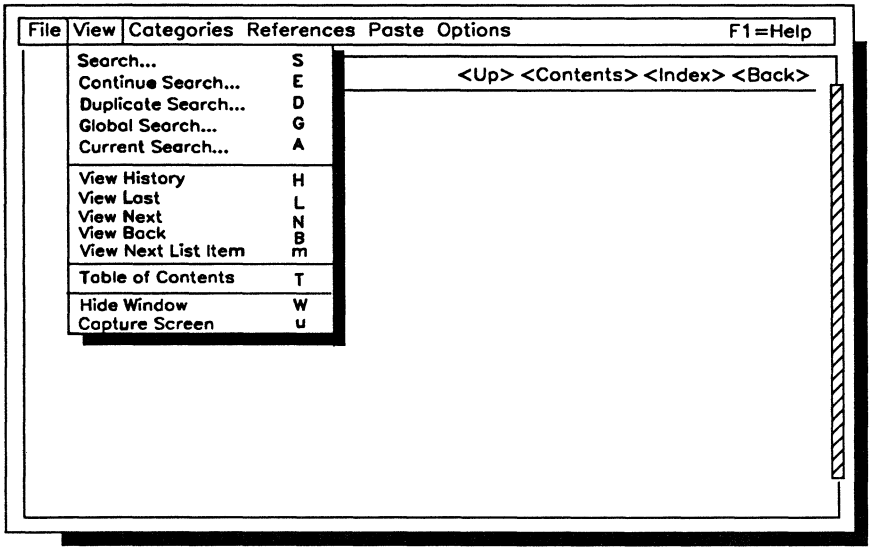


Figure 4-5. View Commands

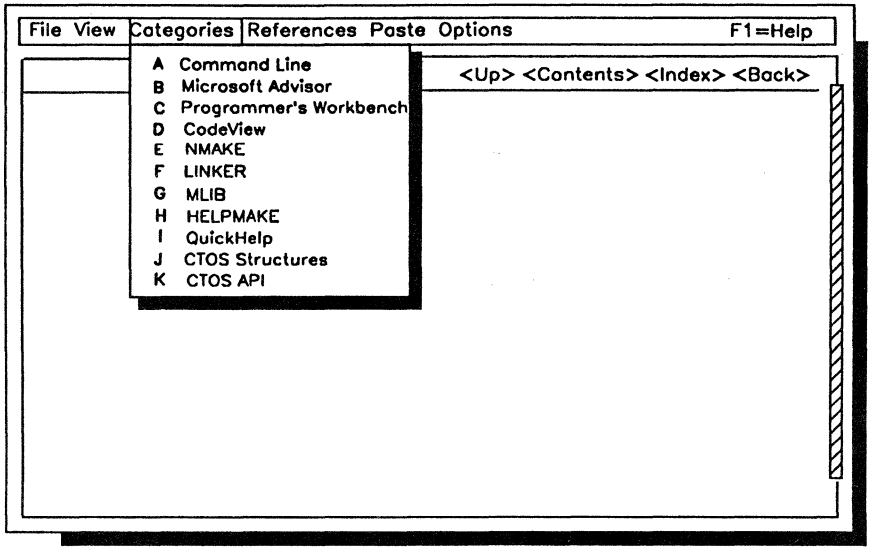


Figure 4-6. Categories Commands

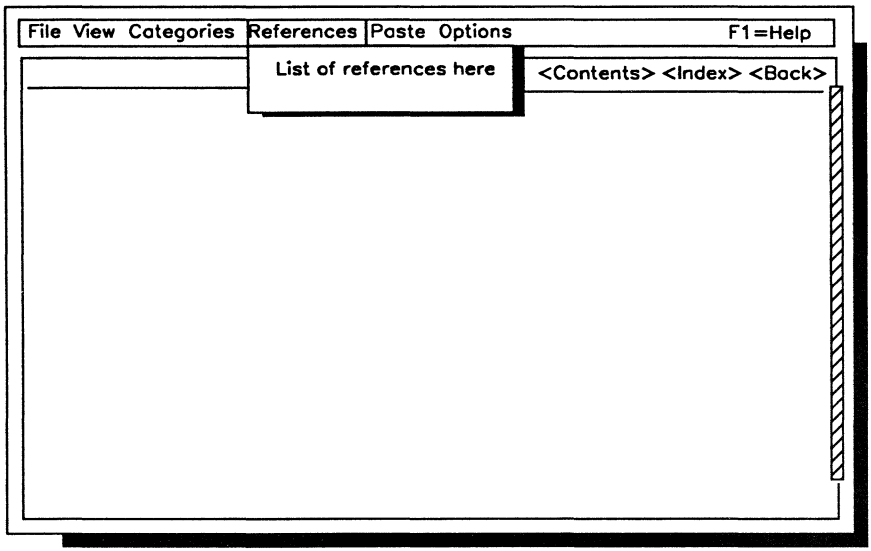


Figure 4-7. References Commands

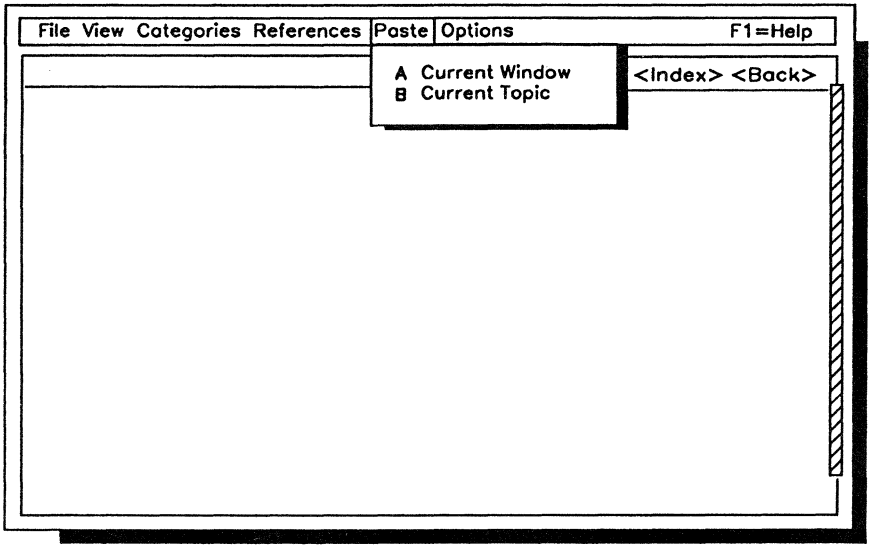


Figure 4-8. Paste Commands

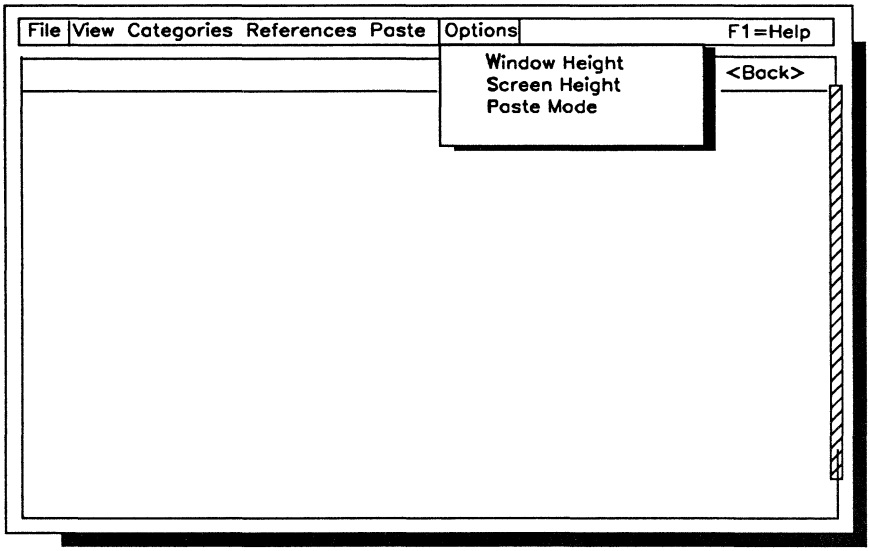


Figure 4-9. Options Commands

To execute a command from an opened menu, choose one of these options:

- Click on the command.
- Highlight the command with arrow keys, and press **GO**.
- Press the highlighted letter to the left of or inside the command.

  For example, to execute the List Databases command from the File menu, press **L**.

  Note: Letters to the left of commands or highlighted inside command names execute commands after menus appear. Letters to the right of commands are called hot keys, and execute commands anytime, anywhere in QuickHelp.

To execute a command from an unopened menu:

- Press the hot key that appears to the right of the command. Note that not all commands have hot keys.

  For example, to execute the File menu command **Print Current Window** from anywhere in QuickHelp, press =.

To display information about a menu command, either:

- Click Right.
- Highlight the command, and press **F1**.

Dialog Boxes

Notice that ellipsis appear next to some commands (. . .). These marks identify commands that, when selected, prompt you for more information with dialog boxes. For example, the box shown in Figure 4-10 appears when selecting the Search command from the File menu.

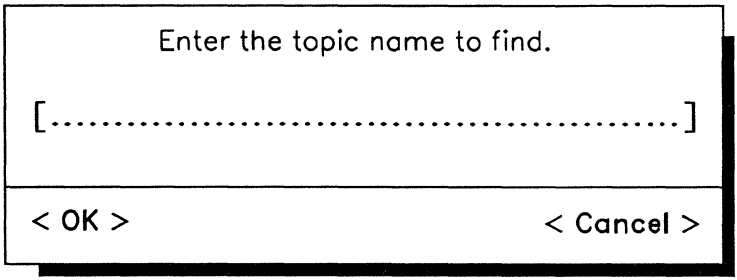


Figure 4-10. Dialog Box for the Search Command

To complete a dialog box, enter the required information, and then either click on **<OK>** or press **GO**.

Note: *The Search command offers a very convenient way to quickly find information about specific topics. For example, to display the syntax and #include file for printf, enter printf and press GO.*

Navigating through Help Screens

To navigate through Help screens in QuickHelp, you can use commands from the View pull-down menu, use hyperlink commands, and hyperlink topics. To navigate through Help screens in the Advisor, you use hyperlink commands and topics only.

This subsection covers the following information:

- Navigating with QuickHelp Pull-Down Menus
- Navigating with Hyperlink Commands
- Displaying Hyperlink Topics

Navigating with QuickHelp Pull-Down Menus

A series of commands in QuickHelp's View menu allow you to selectively display topics. These commands include the following:

Command	Description
View History	Lists all topics recently displayed. To return to a listed topic, click on the topic.
View Last	Displays the last topic.
View Back	Moves backward one topic in the Help file.
View Next	Displays the next topic in the Help file.

Navigating with Hyperlink Commands

For both QuickHelp and the Advisor, <buttons> appear at the top right corner of the Help windows, which are useful for moving through the Help system. To execute these commands, click left once. Table 7-1 describes command functionality.

Table 4-1. Hyperlink Commands

Button	Description
< Up >	Moves upward in the hierarchy of Help screens. Since information is logically ordered, moving from general to specific, this command is useful for moving up the information tree.
< Contents >	Displays the global contents screen. This command always returns you to a known point in the online Help system hierarchy.
< Index >	Displays the global index list. Selecting an item from the list displays the index for that category.
< Back >	Moves you to the last online Help screen you saw.

Displaying Hyperlink Topics

Hyperlink topics are cross-references that connect related information, and are used by both QuickHelp and the Advisor. Hyperlink topics appear in brackets < > called buttons just like hyperlink commands.

To display hyperlink topics, choose one of these options:

- Click left twice.
- Click right once.
- Highlight it and press **GO**.

  To move the highlight (reverse video) from button to button, press **TAB** to go forward, press **SHIFT-TAB** to go backwards.

Hyperlink topics also exist in an unmarked form. An unmarked hyperlink is not delimited with < >. Unmarked hyperlink topics are generally C keywords, library functions, constants defined by CTOS Microsoft C, and similar identifiers.

Unmarked hyperlinks also exist in PWB and the CTOS Microsoft CodeView Debugger. Additionally, PWB recognizes elements of the C language as hyperlink topics in your source code, and the CodeView debugger does the same from active windows. For example, if your source code contains the printf function, you can display printf Help information directly from your source code in PWB. Table 4-2 shows the different ways to display unmarked hyperlink topics.

Table 4-2. Ways to Display Unmarked Hyperlink Topics

	Click Left Twice	Click Right Once	Cursor & F1
QuickHelp	✔	✔	✔
PWB Advisor	✔	✔	✔
PWB Source Code		✔	✔
CodeView Advisor	✔	✔	✔
CodeView Windows			✔

For example, to display syntax and an example for printf from source code in PWB, position the mouse on printf and click right once.

Copying and Pasting

You can copy and paste Help information from both QuickHelp and the Advisor in PWB. This process works differently for each utility, however, so this subsection is divided into two parts:

- Programmer's WorkBench
- QuickHelp

Programmer's WorkBench

In PWB you can copy Help text directly to your editing window. For instance, if you found an example of a little-used function that you needed to use, you can copy the example into your source code and then modify it.

To copy and paste using the mouse, follow these steps:

1. Locate the text to copy in Help.
2. Highlight the text to copy (click left, drag, and release).
3. Click on **Edit, Copy**.

   The highlight disappears.
4. Click the mouse in your edit file (source code) at the place to enter text.
5. Click on **Edit, Paste**.

   The text appears in your edit file.

QuickHelp

From QuickHelp, you can copy text that appears in a window and paste it to another file. For example, you can copy and paste a function example to Editor files, which you can then incorporate into your code.

Note: *If you run QuickHelp and the Editor in Context Manager partitions, you can also copy and paste selectively using ACTION-C and ACTION-P.*

To copy and paste using the mouse, follow these steps:

1. Click on **File, Rename Paste File.**
2. Type the full file specification of the target file.

   Be sure to specify a new file when pasting, since by default, QuickHelp overwrites text in existing files. (The Paste mode command in the Options menu allows you to toggle between overwrite and append.)
3. Click on **< OK >.**
4. Click on **Paste.**
5. Click on either **Current Window** or **Current Topic.**

   The Paste Operation Completed dialog box appears.
6. Click on **< OK >.**

To copy and paste using the keyboard, follow these steps:

1. Press **F** to open the File menu.
2. Press **R** to open the Rename Paste File window.
3. Type the full file specification of the target file.

   Be sure to specify a new file when you paste since, by default, QuickHelp overwrites text in an existing file. (The Paste mode command in the Options menu allows you to toggle between overwrite and append.)
4. Press **GO.**
5. Press **P** to open the Paste menu.
6. Press arrow keys to highlight either the **Current Window** or the **Current Topic.**
7. Press **GO.**

   The Paste Operation Completed dialog box appears.
8. Press **GO.**

To access the copied Help information, follow this procedure:

1. Switch into another partition. (For example, switch to the Editor.)
2. Open the target file.
3. MARK, BOUND, COPY or MOVE the information into your source file.

Displaying Keyword Information

Keywords can be classified as operators, symbolic constants, and library functions. The following procedure applies to both QuickHelp and the Advisor.

To display information about a keyword:

1. Display the appropriate index for the keyword, a sample of which is shown in Figure 4-11.
2. Click on the keyword's first letter from the index.
3. Scroll down the list of entries and click on the topic's hyperlink.

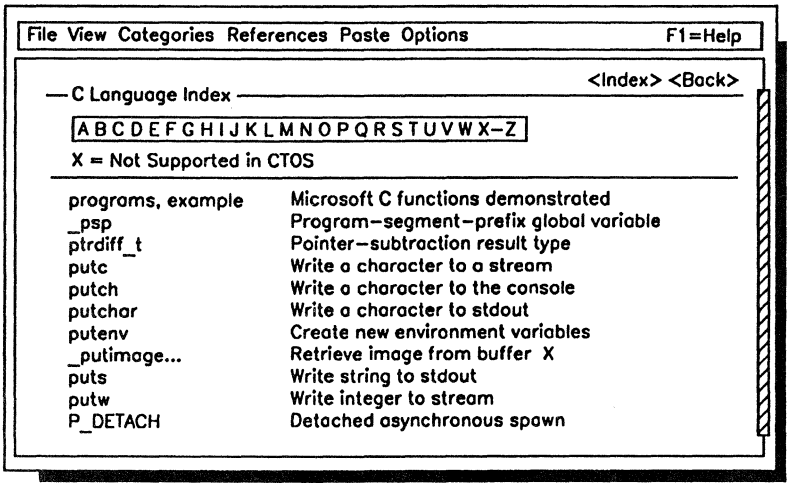

File View Categories References Paste Options F1=Help
<Index> <Back>
C Language Index
A B C D E F G H I J K L M N O P Q R S T U V W X–Z
X = Not Supported in CTOS
programs, example Microsoft C functions demonstrated
_psp Program–segment–prefix global variable
ptrdiff_t Pointer–subtraction result type
putc Write a character to a stream
putch Write a character to the console
putchar Write a character to stdout
putenv Create new environment variables
_putimage... Retrieve image from buffer X
puts Write string to stdout
putw Write integer to stream
P_DETACH Detached asynchronous spawn

Figure 4-11. Sample Index for Keywords

In QuickHelp, if you know the exact name of a keyword it is often faster to select **View**, **Search**, type the name of the keyword in the dialog box, and press **GO**. To display the Search menu directly, press **S**.

When information appears in the Help window, two additional hyperlink buttons also appear:

Button	Description
< Description >	Provides a detailed explanation of the function. When the information appears, the button changes to **< Summary >**. To return to summary information, click on this button.
< Example >	Displays C source code that shows how to use the function.

Displaying Error Code Information

QuickHelp and the PWB Advisor provide information about compiler and linker error messages.

To display error messages with the keyboard:

1. Press **S**.

   The Search dialog box appears.

2. Enter the error number, and press **GO**.

   An explanation appears.

To display error messages with the mouse:

1. Click on **View**, **Search**.

   The Search dialog box appears.

2. Enter the error number, and click on **< OK >**.

   An explanation appears.

An alternative to the above methods is to select **< Contents >**, **< Errors Help>**.

Printing QuickHelp Information

From QuickHelp, you can print information that appears on one screen, and print all information about a topic.

To print Help information with a mouse, follow this procedure:

1. Click on **File**.
2. Choose one of these options:
   - To print information that appears on the screen only, click on **Print Current Window**.
   - To print information about the topic, click on **Print Current Topic**.

   A dialog box appears, prompting you to enter a printer or device name.
3. Type the name of a printer in square brackets (for example, [Spike]), and click on **<OK>**.

   A dialog box appears, prompting you to enter the number of lines per page to print.
4. Either accept the default (66) or enter a new number, and click on **<OK>**.

   The information appears at the printer.

To print Help information with keystrokes, follow this procedure:

1. Press **F** to pull down the File menu.
2. Choose one of these options:
   - To print information that appears on the screen only, press **W**.
   - To print information about the topic, press **T**.

   A dialog box appears, prompting you to enter a printer or device name.

3. Type the name of a printer in square brackets (for example, [Spike]), and press **GO**.

   A dialog box appears, prompting you to enter the number of lines per page to print.

4. Either accept the default (66) or enter a new number, and press **GO**.

   The information appears at the printer.

Configuring a Context Manager Partition

Running QuickHelp under Context Manager is a very efficient way to use it. To run QuickHelp under Context Manager, first configure a partition, then start Context Manager.

To configure a partition, follow this procedure:

1. On the Executive command line, type **CM E** and press **GO**.

   Context Manager field entries appear.

   Note: *If your CM config file is not the default, press RETURN before GO, and enter your config file.*

2. Complete the field entries using the following example:

```
Command Name            QuickHelp
Run file name           [Sys]<msc>QH.Run
Memory required         <500
Abbreviation            QHelp
[Function key (1..10)]  2
[Command case]          00
[Volume]                Sys
[Needs Exec screen?]    n
```

3. Press **FINISH, GO**.

In the above example, QuickHelp automatically opens .Hlp files in the path [Sys]<msc>, assuming environment variables are not specified.

This Page is Blank

Do Not Print

Section 5
Building Applications

This section contains information on building CTOS applications with CTOS Microsoft C. It focuses on command-line utilities used to create run files--the compiler and linker--and describes the object modules and libraries necessary to build run files. This section also contains reference material on makefiles and programming with CTOS Microsoft C.

This section covers the following topics:

- Using the Compiler
- Using the Linker
- Startup Object Files and Heap Architecture
- Using the Environment Service
- Notes on Using Makefiles
- Programming Notes

For a list of all command-line options for CTOS Microsoft C utilities, refer to Appendix B, Command Reference.

Using the Compiler

The CL utility compiles one or more C source files, and then chains to MLINK, an interface to the CTOS Linker, which automatically links the object modules to create run files.

This subsection covers the following topics:

- CL Command Form
- Notes on Command Line Options
- CL's MASM Successor Call
- Compiling for Overlays
- Overlay Incompatibilities
- Syntax Errors

CL Command Form

The compiler command and parameters take this form:

```
CL
   [options] source_files... [libraries link-options]
```

Many command line */options* exist that you can use to control the compiler. If you specify multiple source files, separate them with spaces. To specify [*libraries line-options*], use the /link option first, then list MLINK options and libraries, other than the default libraries needed to produce your run file.

CL automatically chains to MLINK to create run files. For details about MLINK and how to control linking the CL /link option, refer to the heading Using the Linker. For a list of all *options*, refer to Appendix B, Command Summaries.

Notes on Command Line Options

If you do not specify a memory model, the compiler defaults to small model:

```
/AS
```

To stop the compiler from automatically linking with MLINK, specify this option:

```
/c
```

The C compiler may generate valid but different object modules, depending on the amount of memory available for the optimization process. To ensure identical results from multiple compilations, be sure to use the same partition size.

If you set global optimizations /Ox, /Oe, /Ol, or /Og on code that uses functions setjmp or longjmp, the compiler may generate incorrect code. To use these functions and ensure that the compiler generates correct code, either compile without these options, or use the optimize pragma to turn off /Oe, /Ol, and /Og in functions that contain setjmp and longjmp as follows:

```
#pragma optimize ( "elg",off )

        . . . {function containing setjmp or longjmp}

#pragma optimize ( "",on )
```

CL's MASM Successor Call

If you list files with extension .Asm on the CL command line, CL chains to the CTOS Microsoft Macro Assembler (MASM), which assembles the files.

Compiling for Overlays

To produce overlay-compatible code that conforms to CTOS Virtual Code Segment Management facility requirements, compile with these switches:

```
/AL /GY
```

where /GY combines /r /t /Or. Table 5-1 lists and describes these options.

Table 5-1. Compiler Options Used for Overlays

Option	Description
/AL	Compiles with Large model to provide far code pointers. The customized memory model option /Alxx is sufficient for this purpose.
/GY	Generates overlay-compatible code, which conforms to requirements established by the CTOS Virtual Code Segment Management facility. Combines /r, /t, and /Or, and disables /Oc, the local common expression optimization.
/r	Suppresses register variables. Ignores the register keyword and suppresses SI and DI register use. (SI and DI usage is not eliminated for string instructions emitted for mem* and str* intrinsics, and for multibyte initializations.) Disables /Oc. Incompatible with /Oe and /Og.
/t	All static functions become public.
/Or	Single function exit (return). Disables in-line returns from functions.

Overlay Incompatibilities

You cannot overlay C Library or Floating-Point Library modules.

/GY is incompatible with the options shown in Table 5-2.

Table 5-2. CL Compiler Options Incompatible with Overlays

/As	Sets up a customized memory model with small (Near) code pointers. /GY requires far code pointers.
/Oe	Enables global register allocation.
/Og	Enables global optimizations and global common expressions.
/Oy	Removes stack frame; suppresses stack frame generation for functions without arguments and automatic storage

/GY should not be used with intrinsic forms of mem* and str* functions:

```
memcmp
memcpy
memset
strcat
strcmp
strcpy
strlen
strset
```

Syntax Errors

Some combinations of syntax errors in a C source file can cause the C compiler to terminate with a general protection fault (GPF):

```
your_source.c(nnn): fatal error C1131:  GPF in C1 at
sel:off
```

Where nnn is the source line number and sel:off is the execution address in C1.Run at the time the GPF occurred.

Should you encounter a C1131 error, correct the syntax errors already reported by the compiler and recompile.

Using the Linker

CTOS Microsoft C provides the CTOS Linker (MCLinker.Run) and two commands that execute it, MLINK and MCBIND. MLINK provides a command line option interface ; MCBIND provides the standard CTOS command form. MCBIND also includes a new field to specify building a .Cdv file, necessary to use the CTOS Microsoft CodeView debugger.

To use MLINK, execute CL. MLINK starts automatically when CL finishes, receives the compiler information about object modules and file names, uses libraries that correspond to the compiled memory model, and creates the run file.

You can also use MLINK independently if you compile with option /c, which disables automatic linking. MLINK provides command line options for all CTOS Linker command form fields and config file parameters. For a complete list of command line options, refer to Appendix B, Command Reference.

To use MCBIND, first compile with option /c, and then use the MCBIND command form to specify startup object modules and libraries. You can also specify optional flavor object modules. The startup and flavor modules control the types of run files created and affect heap architecture.

Note: *MLINK uses default startup object modules and libraries, which you can override with options from CL. For a complete description of how startup and flavor object modules affect heap architecture, refer to the subsection Startup Object Modules and Heap Architecture.*

This subsection covers these topics:

- Using MLINK
- Using MCBIND
- Cross References between MLINK and MCBIND Options
- Creating a Typical Run File
- Startup and Flavor Object Modules
- Specifying Standard Libraries
- Linking a Minimum Size Run File
- Linking Floating-Point Support
- Creating, Naming, and Using Combined Libraries

Using MLINK

MLINK is a command line interface to the CTOS Linker. It defaults to automatic linking after compiling with CL. You can also use it separately if you suppress automatic linking with the CL option /c.

This subsection covers these topics:

- Command Overview
- Controlling Automatic Linking
- Using an MLINK Response File
- MLINK Defaults

Command Overview

MLINK uses the following syntax:

```
MLINK
objfiles [,[runfile] [,[mapfile] [,[libraries]]]] [options] [;]
```

Table 5-3 lists and describes the fields.

Table 5-3. MLINK Command Fields

Field	Action/Explanation
objfiles	Name or names of object files. Separate multiple file names with spaces. If you do not append the .Obj extension, MLINK appends one for you.
runfile	Name of the run file to create. If you do not append the .Run extension, MLINK appends it. If you do not specify this option, MLINK names the file *first_obj*.Run. Also, if you do not specify this option but specify *mapfile* or *libraries*, the comma is required.
mapfile	Name of the file to receive the map listing. If you do not append the .Map extension, the CTOS Linker appends it. If you do not specify this option, the CTOS Linker names the file *first_obj*.Map. Also, if you do not specify this option but specify *runfile* or *libraries* , the comma is required.
libraries	Path and name of libraries needed for linking. If you do not append the .Lib extension, MLINK appends it.

continued

Table 5-3. MLINK Command Fields (cont.)

Field	Action/Explanation
options	Linker options. Use any of the command line options listed for MLINK in Appendix B, Command Reference. Options cross referenced to MCBIND and LinkerConfig.Sys fields appear in tables in this subsection. *options* can appear anywhere on the command line; they must be separated with spaces. Option *values*, if supplied, must be separated from the option (and other values) with a colon (:). For example, /RU:*mode*, where *mode* is the option value.
;	Indicates the end of command line options. You can use the semicolon anywhere on the command line. All options to the right of the semicolon, if supplied, are ignored.

Note: *MLINK does not append an extension to a filename that has a period (.) for a last character, and does not consider a last-character (trailing) period as part of the filename.*

Controlling Automatic Linking

When used automatically from CL with defaults, MLINK receives all *objfiles* from CL, names the *runfile* to *first_obj*.Run, lets the CTOS Linker name *mapfile* to *first_obj*.Map, and uses the libraries that conform to the compiled memory model.

With CL compiler switches, you can override the defaults and control MLINK. With /Fe you can name the *runfile*, with /Fm you can name the *mapfile*, and with /link send to MLINK any combination of *libraries* and MLINK *options*. If you use, /link must be the last input on the CL command line. You must separate MLINK options and library names with spaces.

All compiler switches except /link must precede the source file names to which they apply. For example,

```
CL
  [Args]  Opt1 source1.c Opt2 source2.c /link
```

will compile source1.c with Opt1; however, source2.c will compile with both Opt1 and Opt2 .

To show how to control MLINK from CL, the next example shows how to use compiler options. It shows how to create a file that can be inspected with the Source Browser in the Programmer's WorkBench, and debugged with the CTOS Microsoft CodeView debugger. It also shows how to invoke MASM to assemble assembly files, and pass object files and options to MLINK.

```
CL
   /AL /Od /Zi /FR  Io1.c Io2.Obj Io3.Asm
```

In this example, CL:

- Compiles with a large model (/AL).
- Compiles without optimization (/Od).
- Generates CodeView information (/Zi).
- Generates a Browser file Io1.Sbr (/FR).

CL also works with three different file types: C source, object module, and assembly. In this case, CL:

- Compiles Io1.c.
- Skips Io2.Obj.
- Calls MASM to assemble Io3.Asm.

When finished compiling, CL passes to MLINK:

- All three object module names.
- The /link option /CO (CodeView option), which occurs automatically with compiler switch /Zi.

MLINK then chains to the CTOS Linker, which creates Io1.Run and Io1.Cdv (the CTOS Microsoft CodeView debugger file).

Notes: *The above example assumes that all libraries, header files, and source files are in the current working directory. If you keep these files in separate directories, you can direct search paths to them with environment variables LIB, INCLUDES, and PATH.*

To create the final Io1.Bsc Browser file from the command line, the file required to use the Browser, you must run PWBRMAKE on Io1.Sbr. When created from the Programmer's WorkBench (PWB), the Browser file Io1.Sbr automatically converts to Io1.Bsc. For details, refer to Section 3, Using the Programmer's WorkBench.

You can also set MLINK */options* with the Environment Service variable MLINK. You can specify this variable two ways. One way is to create this entry in [Sys]<Sys>EnvironmentConfig.Sys:

```
:MLINK: /options
```

separating *options* with spaces. The other way is to issue this command:

```
Environment Set

  MLINK=/options
```

Either way, separate all options with spaces. You can also specify libraries with the LIB environment variable.

For a brief description on how to use Environment Service, refer to Using Environment Service in this section. For a list of environment variables recognized by MLINK, refer to Appendix B, Command Reference. For a complete description of how to use Environment Service, refer to your *CTOS Environment Service Administration and Programming Guide.*

Note: *MLINK defaults to using the standard startup object module, MC0.Obj, described in this section.*

Using an MLINK Response File

If you use MLINK from the Executive, you can specify file names and options in an MLINK Response File (MRF) using the following syntax:

```
MLINK
     ~MRFfilename
```

Configure *MRFilename* this way:

```
object files
[runfile]
[mapfile]
[libraries]
```

You must specify at least one object file name. If you omit any group, leave a blank line. You can enter *options* anywhere, but they must be separated with spaces. If MLINK finds a semicolon, it supplies default file names for all files not yet specified, and ignores the rest of the file. To specify object files or libraries on multiple lines, end each line except the last with a plus sign(+). For example,

```
Object1.Obj Object2.Obj +
Object3.Obj
Object1.RUN

Library1.Lib Library2. +
Library3
```

To specify file names without extensions, you must end the file name with a period. MLINK automatically removes the period (.). To have MLINK automatically add the common extensions for you, do not use the period or extension. For example, in the above sample, MLINK treats `Library2.` as `Library2`, and treats `Library3` as `Library3.Lib`. This rule applies to all file names in an MRF.

MLINK Defaults

Table 5-4 lists default MLINK options as they correlate to the MCBIND command form.

Table 5-4. MLINK Defaults Correlated to MCBIND Fields

MCBIND Field	Correlating MLINK Default
Object modules	MC0.Obj myprog.Obj
Run file	myprog.Run
[Map file]	myprog.Map (CTOS Linker names it)
[Publics?]	
[Line numbers?]	
[Stack, Heap size]	
[Max array, data]	0 0
[Min array, data]	0 0
[Runfile mode]	Protected
[Version]	mm/dd/yy hh.mm
[Libraries]	LLIBCP.LIB LIBCEXT.LIB EM.LIB LLIBFP.LIB
	When used automatically from CL, MLINK uses the appropriate libraries according to the memory model used with CL
[DS allocation?]	No
[Symbol file]	myprog.Sym (CTOS Linker names it)

continued

Table 5-4. MLINK Defaults Correlated to MCBIND Fields (cont.)

MCBIND Field	Correlating MLINK Default
[Copyright notice?]	
[File to append]	
[Linker Configfile]	
[Source Debugger]	

Note: If you use MLINK from the Executive command line, [libraries] defaults to large libraries and floating-point emulation. If you are familiar with CTOS Microsoft 6.0, note that for 6.1 floating-point startup modules EM.Obj and 87.Obj are replaced with EM.LIB and 87.LIB. This change allows automatic linking of floating-point emulation if your programs use it.

Using MCBIND

When compiling and linking in separate steps (when specifying option /c for CL), MCBIND, as shown in Figure 5-1, is the CTOS command form used to link object modules. The command form looks and works the same way as previous CTOS Linker command forms except for the final two options: **[Linker Configfile]** and **[Source Debugger]**.

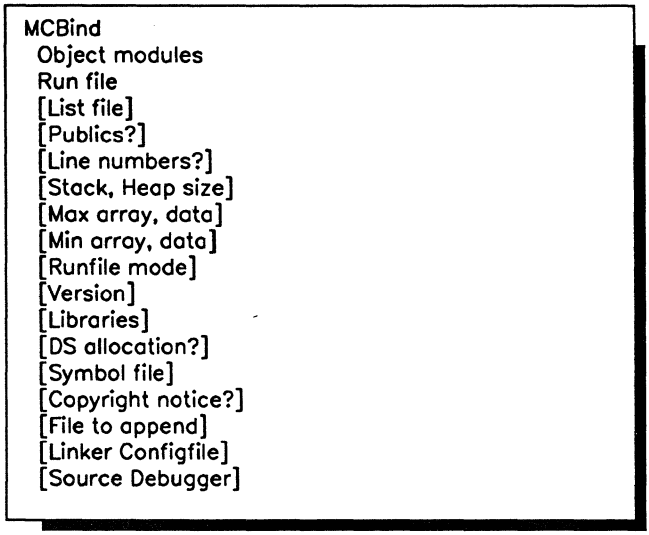


Figure 5-1. MCBIND Command Form

To use MCBIND, you must complete the fields **Object Modules** and **[Libraries]** with startup modules and libraries supplied with CTOS Microsoft C, described in this subsection.

To use the CodeView debugger, you must compile with option /Zi, and enter **CodeView** in the **[Source Debugger]** field. This linker option builds a special file required by the CodeView debugger, *runfile*.Cdv. For details on how to use CodeView, refer to Section 13, Debugging with CodeView.

Cross References between MLINK and MCBIND Options

MLINK's command line options duplicate all MCBIND fields, and all Linker config file options that have no MCBIND command form equivalents.

The following tables show the relationships between MLINK options and MCBIND command-form fields, and between MLINK options and linker config options. For a complete list of options, refer to MLINK in Appendix B, Command Reference.

Table 5-5 lists MLINK switches that duplicate MCBIND command-form field options.

Table 5-5. Correlating MCBIND and MLINK Options

MCBIND Option	Correlating MLINK Command Line Options
Object modules	MC0.Obj or /NOR[UNTIME] (or /MC0N[RT])
	or /NOST[ACKSHARE] (or/MC0H[EAP])
	/MI[NRUNTIME], /RE[ALHUGE] (flavor objects)
Run file	[,[*runfile*]] else basename.Run
[Map file]	[,[*mapfile*]] or /NOM[AP]
[Publics?]	/PU[BLICS]
[Line numbers?]	/LINE[NUMBERS]
[Stack, Heap size]	/ST[ACK]:*stacksize*, /HEA[P]:*heapsize* or :*MAXVAL*
[Max array, data]	/MAXM[EM]:n0:n1 (default: 0 0)
[Min array, data]	/MINM[EM]:n0:n1 (default: 0 0)
[Run file mode]	/RU[NFILEMODE]:*mode* (default: Protected)
[Version]	/VE[RSION]:*version* or /NOV[ERSION]

continued

Table 5-5. Correlating MCBIND and MLINK Options (cont.)

MCBIND Option	Correlating MLINK Command Line Options
[Libraries]	/NOD:*libname* [:*libname*] . . . or /NOD for none
	/FP8[7], /FPE[M], /FPA[LT, or NOFL[OAT]
	/AS, /AC, /AM, /AL (for C and FP library model) - default: /AL
	/LIBR[EF]
[DS allocation?]	/DS[ALLOCATE] (default: NO)
[Symbol file]	/SY[MBOLFILE]:*symbolfile* or /NOSY[M]
[Copyright notice?]	/CR[IGHT]
[File to append]	/AP[PENDFILE]:*appendfile*
[Linker Configfile]	/FI[LECONFIG]:*config file*
[Source Debugger]	/CO[DEVIEW]

An additional option, V8, specifies the version 8 run file type. Version 6 is the default.

Notes: */NOD[EFAULTLIBRARYS]:libname requires a libname from the default library list: LIBCP.LIB, LIBCEXT.LIB, CTOS.Lib, CTOSToolkit.Lib. /NOD without libname suppresses all default libraries.*

To display a table similar to Table 5-5 on your screen without linking, execute MLINK with the /SHOW option. To display the same table as you link, execute MLINK with the /INFO option.

Table 5-6 lists MLINK switches that duplicate CTOS Linker config file options.

Table 5-6. Correlating CTOS Linker Config File and MLINK Options

Config File Option	Correlating MLINK Option
:CaseSensitive:Yes	/NOI[GNORECASE]
:CharacterCodeSet:*codeset*	/CH[ARSET]:*codeset*
:ClassOrder:*classlist*	/CL[ASSORDER]:*classlist*
:CopyRight:*string*	/CS[TRING]:*string*
:Details:Yes	/DE[TAILS]
:DGroupHeapSize:	/HEA[P]:*heapsize*
:FarCallTranslation:Yes	/FAR[CALLTRANSLATION]
:NoFarCallTranslation:No	/NOFA[RCALLTRANSLATE]
:NonContiguousGroupsOK:Yes	/NONC[ONTIGUOUSOK]
:PackCode:Yes	/PACKC[ODE] [:*size*]
:SuppressWarnings:Yes	/NOW[ARNINGS]
:Undefined:*symbolname*	/U[NDEFINED]:*symbolname*
:MaxCodeViewCode: *Sector_Count*	/XCVC[ODE]:*number*
:MaxCodeViewLines: *Sector_Count*	/XCVL[INES]:*number*
:MaxExportData:*Sector_Count*	/XEX[PORTDATA]:*number*
:MaxImportData:*Sector_Count*	/XIM[PORTDATA]:*number*
:MaxRgIdct:*Sector_Count*	/XID[CT]:*number*

continued

Table 5-6. Correlating CTOS Linker Config File and MLINK Options (cont.)

Config File Option	Correlating MLINK Option
:MaxRgPdhCG:*Sector_Count*	/XPD[HCG]:*number*
:MaxRgRlePStub:*Sector_Count*	/XRG[RLEPSTUB]:*number*
:MaxRgRle:*Sector_Count*	/XRL[E]:*number*
:MaxRgRqlable:*Sector_Count*	/XRQ[LABLE]:*number*
:MaxStringsTOC:*Sector_Count*	/XST[RINGSTOC]:*number*
:MaxV8Strings:*Sector_Count*	/XV8[STRINGS]:*number*

Creating a Typical Run File

This subsection shows how to create a run file, first with MCBIND and then with MLINK. Since MLINK provides MCBIND arguments to the CTOS Linker, MCBIND usage is discussed first.

Figure 5-2 shows a sample MCBIND command form that creates Test.Run, a typical CTOS run file that performs some floating-point computation. You can run the resulting version 6 run file on both real and protected mode CTOS systems. The text that follows lists the mandatory fields you must complete, and briefly describes how they are completed in the example.

```
MCBind
  Object modules      MC0.Obj Test.Obj
  Run file            Test.Run
  [List file]
  [Publics?]
  [Line numbers?]
  [Stack, Heap size] 4096
  [Max array, data]  0 0
  [Min array, data]  0 0
  [Runfile mode]     Protected
  [Version]
  [Libraries]        LLIBCP.LIB LIBCEXT.LIB EM.LIB LLIBFP.LIB
  [DS allocation?]   No
  [Symbol file]
  [Copyright notice?]
  [File to append]
  [Linker Configfile]
  [Source Debugger]
```

Figure 5-2. Sample MCBIND Link

The example illustrates information entered into four mandatory fields and four optional fields. The four mandatory fields are:

```
Object Modules
Run file
[Libraries]
[DS allocation]
```

The **Object Modules** field illustrates where to place startup object modules: first, before any other module. MCBIND links the startup module with Test.Obj to create Test.Run, specified in **Run file**, the second mandatory field.

CTOS Microsoft C supplies three startup object modules and two flavor object modules. You must specify one startup object module only, and optionally specify one flavor module only.

The **[Libraries]** field reveals two mandatory libraries, and two optional libraries. The first two libraries, LLIBCP.LIB and LIBCEXT.LIB, are mandatory. LLIBCP.LIB is the large C library (a form of *x*LIBCP.LIB). LIBCEXT.LIB, the C Library Extension, maps C Library functionality to CTOS.calls. The last two libraries, EM.LIB and LLIBFP.LIB, are optional but are required for floating-point emulation. EM.LIB contains the code for the 80x87 emulator; LLIBFP.LIB contains the large model floating-point functions.

The final mandatory field is **[DS Allocation?]**. You must always specify **N** or **No**. The DGroup organization imposed for DS allocation is incompatible with the organization required by CTOS Microsoft C.

To create the run file illustrated in Figure 5-2 with MLINK, compile with these switches:

```
CL

   /options  test.Obj  /link /ST:4096
```

Note: *Instead of /link /ST:4096, you could specify CL option /F1000 before test.Obj.*

Startup and Flavor Object Modules

CTOS Microsoft C contains three startup object modules and two flavor object modules. The startup modules define segment ordering and near heap architecture for your C program. The flavor modules contain special code to implement features that are not required by all C programs: minimum-sized run files, and a special case providing _huge memory support for real mode programs.

Tables 5-7 and 5-8 list the startup and flavor modules, and include brief descriptions. For a complete description of how and why to use these modules, refer to the heading Startup Object Modules and Heap Architecture.

Table 5-7. Startup Object Modules

Module	Description
MC0.Obj	Standard C startup module.
MC0HEAP.Obj	Primary Near Heap only.
MC0NRT.Obj	No run time support (no C Library support).

Table 5-8. Flavor Object Modules

Module	Description
MIN.Obj	No files or memory management.
REALHUGE.Obj	Uses real mode _huge memory addressing.

Specifying Standard Libraries

CTOS Microsoft C does not include or create the Combined Libraries familiar to DOS/OS2 Microsoft C programmers, nor is the CTOS Linker sensitive to the Default Libraries record the C compiler places in the object files. Consequently, a description of the standard library files provided with CTOS Microsoft C is presented here, as shown in Table 5-9.

Table 5-9. Standard Libraries

Library	Contents
*x*LIBCP.LIB	Standard C Libraries (Required).
x = S, C, M, or L	These libraries are essentially the same as Microsoft's protected mode versions.
LIBCEXT.LIB	Interface Between C Library functions and CTOS (Required).
	Must always follow *x*LIBCP.LIB. Direct calls to functions in Libext.Lib by user programs are not supported.
LIBH.LIB	No C Run-Time Library
	Contains compiler long-helper functions, and the model-independent string and memory routines. Use this library with startup module MC0NRT.Obj only. Incompatible with all other libraries: specify this library only.
EM.LIB	80x87 Emulator Library.
	Contains the floating-point 80x87 emulator, and must accompany the proper floating-point library for the memory model, *x*LIBFP.LIB.
87.LIB	80x87 Coprocessor Library.
	Contains the 80x87 coprocessor code, and must accompany the proper floating-point library for the memory model, xLIBFP.LIB.

continued

Table 5-9. Standard Libraries (cont.)

Library	Contents
*x*LIBFP.LIB	Floating-point Libraries.
x = S, C, M, or L	Provided in the four memory model versions. To use this library, you must also specify either EM.LIB, for floating-point emulation, or 87.LIB for 80x87 coprocessor only, on the Linker's [Libraries] argument line .
xLIBFA.LIB	Alternate Math Libraries.
x = S, C, M, or L	Provided in the four memory model versions, and must be used when compiling with the /FPa command line option. If used, do not use EM.LIB or 87.LIB.

Linking a Minimum-Size Run File

A small size run file may be produced by eliminating C-file I/O, parameter parsing, and malloc memory management. As a result, a minimum-size run file lacks the stdin, stdout, or stderr streams, and accessing these streams causes an error. In addition, argc and argv are not available to main( ), and the environment segment is not established.

A minimum sized run file eliminates the SAM modules required from CTOS.Lib, and is typically needed when creating a CTOS system service. To retain the reduced size, argument parsing or memory management should be accomplished by direct calls to the appropriate CTOS functions.

To link a minimum run file with MCBIND, specify **MC0.Obj** and **MIN.Obj** in the **Object Modules** field:

```
Object Modules    MC0.Obj  MIN.Obj  myprog.Obj ...
```

To create a GDT Protected run file, specify **GDTProtected** in the **[Run file mode]** field. Because _huge memory support requires use of the LDT, the use of _huge memory is not supported for GDTProtected run files.

To link a minimum, GDT Protected run file with MLINK, compile with these options:

```
CL
   /options  myprog.Obj  /link /NOR /RU:GDTProtected
```

To create the smallest possible run file, which eliminates all C Library support (no run time), use the startup object module MC0NRT.Obj. You must also specify library LIBH.LIB, which contains compiler long-helper functions, and the model-independent string and memory routines. The above example invokes MC0NRT.Obj with the /link /NOR option.

To link the smallest possible run file with MCBIND, specify MC0NRT.Obj in the Object Modules field, and LIBH.LIB in the **[Libraries]** field:

```
Object Modules     MC0NRT.Obj  myprog.Obj ...

[Libraries]        LIBH.LIB
```

Linking Floating-point Support

Two libraries, EM.LIB and 87.LIB, are provided to specify floating-point support. EM.LIB contains code for the 80x87 coprocessor emulator, which provides floating-point support on all workstations with or without an 80x87 coprocessor. 87.LIB contains code for the 80x87 coprocessor, which requires an 80x87 coprocessor.

Both of these libraries must be used with the floating-point library xLIBFP.LIB. If you use the Alternate Math package, xLIBFA.LIB, you must not specify EM.LIB or 87.LIB. For details on using floating-point, refer to Section 9, Controlling Floating-point Math Operations.

Creating and Naming Combined Libraries

If you are used to working with Microsoft's LINK, you may prefer to work with combined libraries, which accommodate the memory model and floating-point requirements for the executable file. Combined libraries are not included with CTOS Microsoft C, but you can make them with MLIB. If you make the libraries, you can specify them in the **[Libraries]** field for MCBIND. The three combined libraries are shown in Table 5-10.

Table 5-10. Combined Libraries

Math Version	Contents	Combined Library
80x87 Emulator	+EM.LIB+*x*LIBCP.LIB+*x*LIBFP.LIB	*x*LIBCE.LIB
In-line 80x87	+87.LIB+*x*LLIBCP.LIB+*x*LIBFP.LIB	*x*LIBC7.LIB
Alternate Math	+*x*LIBCP.LIB+*x*LIBFA.LIB	*x*LIBCA.LIB

To create a combined library, use the MLIB command as follows:

```
MLIB
      Contents ,,Combined_Library
```

For complete MLIB syntax and a list of options, refer to Appendix B.

To use a combined library, enter the combined library name in the MCBIND [Libraries] field. For example, if you create a combined library for the large model emulator, instead of entering:

```
[Libraries]    LLIBCP.LIB LIBCEXT.LIB EM.LIB LIBFP.LIB
```

you could enter:

```
[Libraries]    LLIBCE.LIB
```

Startup Object Modules and Heap Architecture

With different combinations of CTOS Linker options and startup object modules, you can size DGroup and near heap memory, control placement of near heap memory in DGroup, and specify a near heap memory overflow area.

This subsection covers these topics:

- Allocating Near Heap Memory with malloc( ) and _nmalloc( )
- Sizing Near Heap Memory
- Placing Near Heap Memory in DGroup: Three Options
- Sizing and Allocating Memory in Far Heap

Allocating Near Heap Memory with malloc( ) and _nmalloc( )

Memory allocated via calls to malloc( ) for small and medium memory models is located in the near heap. In these memory models, the malloc() function maps to _nmalloc(). In any memory model, explicit calls to _nmalloc() allocate memory from the near heap. The near heap, and thus all memory allocated by _nmalloc(), is located in DGroup, your program's default data segment.

Sizing Near Heap Memory

One way to calculate the maximum amount of near heap memory that your program can allocate is with this formula:

```
Max_Near_Heap = (DGroup size) - (data size) - (stack size)
```

To size the near heap after making this calculation, you can use the Heap Size subparameter in the Linker's **[Stack, Heap size]** field. For example, the entries:

```
[Stack, Heap size]2000 10000
```

specify a 2000-byte stack and a 10,000-byte near heap.

To maximize the near heap size, enter the Heap Size subparameter MAXVAL, as shown in this example:

```
[Stack, Heap size] 2000 MAXVAL
```

Specifying MAXVAL causes the Linker to provide a full 64K DGroup data segment, which maximizes the near heap size. The equivalent MLINK options are:

```
/ST:2000  /HE:MAXVAL
```

Placing Near Heap Memory in DGroup: Three Options

By combining startup modules MC0.Obj or MC0HEAP.Obj with stack and heap size specifications, you can control near heap placement and growth within DGroup. Three placement options are possible:

- Stack sharing
- Primary Area with Stack Sharing Overflow
- Primary Area Only

These options create three different DGroup stack and heap organizations, as shown in Figure 5-3. Beneath each illustration, notice the startup object modules and stack and heap specifications that create each option.

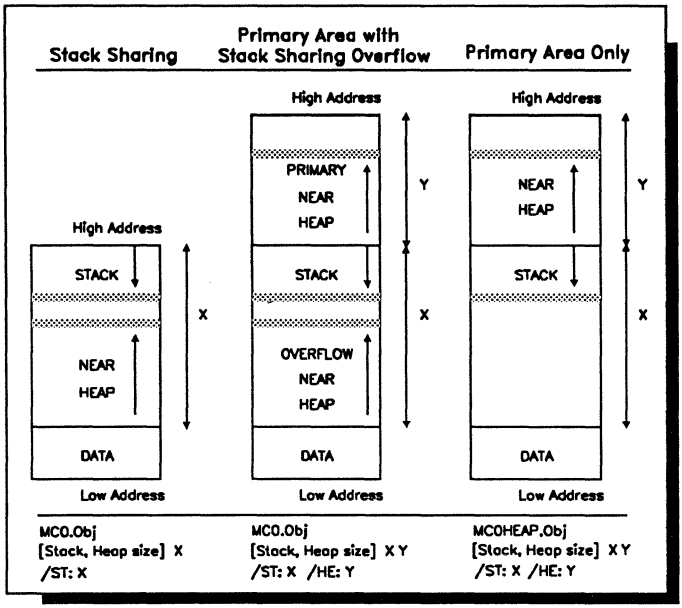


Figure 5-3. Near Heap Memory Organization Options

Stack Sharing

With stack sharing, the near heap and stack share the same memory. The near heap grows from the low-numbered address end of the stack towards the stack pointer. In other words, the stack and near heap grow towards each other. As a result, you must specify a large enough stack size to contain the near heap plus the stack growth expected at run time.

To use this option, use MC0.Obj as your startup object modules (the MLINK default), and either specify an appropriate stack size, or accept the default. Do not specify a heap size.

Primary Area with Stack Sharing Overflow

With this option, the heap is divided into two areas: primary and overflow. The primary near heap and stack do not share the same memory, but the near heap overflow and stack do share memory. The primary near heap now appears at addresses higher than the initial stack pointer. As a result, the primary near heap grows towards the high-address end of DGroup until it reaches the limit (and overflows). Near heap allocation then switches to the low address end of the stack area.

To use this option, use **MC0.Obj** as your startup object module (the MLINK default), and specify both stack size and heap size. The heap size parameter specifies the primary near heap size. To maximize DGroup size and create the largest possible primary near heap, specify MAXVAL for the near heap size rather than a number.

Primary Area Only

With this option, the near heap is restricted to the primary area only, allowing the near heap to be exhausted without consuming the entire stack area, which causes stack overflow. The Primary-Area-Only near heap architecture supports the common Microsoft application program technique of measuring the near heap available at program startup by allocating from the near heap until no more near heap memory is available.

To use this option, use MC0HEAP.Obj as your startup object module (/NOS or /MCOH if using MLINK), and specify both stack size and heap size.

Sizing and Allocating Memory in Far Heap

The following subsections describe how to use the far heap and _huge memory architecture, describe LDT-based segments, and cover these topics:

- Allocating Far Heap Memory with malloc( ), _fmalloc( ), and halloc( )
- LDT-based Segments
- Reserving More Segments with MCMEMORY.Obj
- Linking for Real Mode with _huge Memory

Allocating Far Heap Memory with malloc( ), _fmalloc( ), and halloc( )

Memory allocated via calls to malloc( ) for the compact and large memory models is located in the far heap. In these two memory models, the malloc( ) function maps to _fmalloc( ). In any memory model, explicit calls to _fmalloc( ) allocate memory from the far heap.

By default, the _fmalloc( ) function operates by acquiring 64K segments from your program's partition, and then allocates portions of these segments to satisfy individual _fmalloc( ) calls. As each segment is exhausted, a new one is acquired.

The two variables that affect memory allocation defaults are _amblksiz, initialized to 8K, which controls malloc's heap allocation granularity; and _api_min_alloc, initialized to 0, which controls the granularity of allocation requests to AllocMemorySL. When _api_min_alloc is 0, AllocMemorySLs are for full 64K segments. Setting _api_min_alloc to 1 causes AllocMemorySL's for _amblksiz bytes. You can increase or decrease _amblksiz to suit your application's needs.

Note that with _api_min_alloc set to 1, each allocation for _amblksiz bytes consumes one of the 32 LDTs your program has available by default, making it more likely that all of the LDT's will be consumed. You can increase the number of LDTs by linking in additional copies of MCMEMORY.Obj, each of which provides 32 additional slots.

Setting these allocation variables to new values in your program's code does not affect the initial 64K allocation for argument and environment space. To change the initial allocation behavior, modify the variable definitions in MALLOCMODE.Asm, located in your libraries directory, and link with your updated version of MALLOCMODE.Obj. See the header in the MALLOCMODE.Asm source file for more information.

For _huge memory support, calls to halloc( ) result in the allocation of one or more 64K segments with contiguous selector numbers. The code generated by the C compiler to manipulate _huge memory addresses allows each multi-segment set to behave as a single linear address space.

LDT-based Segments

Both _fmalloc( ) and halloc( ) use a common set of data segments reserved in your program's run file. These segments are made to alias, or overlay, real segments as returned by AllocMemorySL, so that all memory addresses delivered by _fmalloc( ) and halloc( ) are based in the local (LDT range of) selectors.

Reserving More Segments with MCMEMORY.Obj

By default, 33 far heap segments are available for use by _fmalloc( ) and halloc( ), representing roughly two megabytes of total potential allocation. This potential may not be fully achieved, however, due to non-optimal packing, which causes premature segment exhaustion. Non-optimal packing results from the allocation of very large memory blocks with _fmalloc( ), and the use of partial segments for _huge memory by halloc( ).

If your program requires more than the default number of segments, you can provide them with MCBIND by linking one or more copies of MCMEMORY.Obj, specified as:

```
Object modules      MC0.Obj MCMEMORY.Obj myprog.Obj ...
```

To add multiple copies of MCMEMORY.Obj, repeat the filename.

To provide more segments with MLINK, use this option:

```
/SE[GMENTS]:n
```

where *n* is the number of copies of MCMEMORY.Obj to use.

Each copy of MCMEMORY.Obj adds thirty-two segments to the far heap set managed by _fmalloc( ) and halloc( ).

Linking for Real Mode with _huge Memory

The code generated by the C compiler for _huge memory addressing uses constants rather than variables for _huge address arithmetic. As a result, _huge memory support cannot be tailored at run-time for protected vs. real computation.

Therefore, if your program uses _huge memory and must run in both real and protected mode, two run files are required because the default functionality supports protected _huge memory. A second run file is needed to enable real mode '_huge' memory usage.

To link such a run file, first specify MC0.Obj on the Linker's Object Modules argument line, then specify the real mode _huge flavor module, REALHUGE.

Using the Environment Service

CTOS Microsoft C is overpacked with the CTOS Environment Service utility, which allows you to define search paths, storage areas, and default command-line options for various utilities. For example, you can set up CL, the compiler, with a series of switches to execute each time you run CL. If you set up CL this way, you could compile and link by specifying source code files only.

Environment Service includes a series of commands you can use to set, clear, and reload variables. This manual, however, does not fully document Environment Service features and functions. For complete information, refer to your *CTOS Environment Service Administration and Programming Guide.*

To use Environment Service, you must install it with the Executive command Install Environment Service. During installation, it reads the configuration file [Sys]<Sys>EnvironmentConfig.Sys. The following example shows a configuration file.

```
:INCLUDE:[sys]<mscIncludes>
:LIB:[sys]<msclib>
:CL:/Fc /Fs /Al /c /Gs /Zp1
:QH:[sys]<Msc>
:TMP:[!m0]<$>
```

Note that each line has two basic entries: a token on the left and a value on the right. For example, on the first line the token is :INCLUDE: and the value is [sys]<mscIncludes>. This line directs all searches for #include files to the <mscIncludes> directory. On the last line the token is :TMP: and the value is [!m0]<$>, a temporary directory on a ram disk located on a server (an easy way to boost performance). Appendix B, Command Reference, lists and briefly describes the tokens recognized by CTOS Microsoft C utilities, and lists all commands.

Occasionally, you may want to temporarily change values in the configuration file without editing the file, and later reset your system to the values in the file. To change the values without editing, use the command Environment Set. With this command, you must set values with an equal sign. The next example shows how to duplicate the first line of the sample configuration file with Environment Set.

```
Environment Set

    INCLUDE=[sys]<mscIncludes>
```

This example shows how to change the first line of the sample configuration file:

```
Environment Set

    INCLUDE=[sys]<mscIncludesNew>
```

This change takes effect immediately, but disappears if you reboot your machine or issue the command Environment Reload, which reloads all values from the configuration file.

In addition, you can control the size of Environment memory with a Config.sys entry:

:sEnvironment:

If this entry is missing, most systems default to 1024K.

Note: Using the CL and MLINK environment variables can cause unpredictable build behavior under the Programmer's WorkBench (PWB). This is because PWB reads Tools.Ini and Current.Sts to get environment variables, which may be different than those set with Environment Service. Therefore, when building from PWB, you may wish to disable the CL and MLINK environment variables.

Notes on Makefiles

This subsection covers two topics:

- Two Types of Makefiles
- NMAKE with Batch

Two Types of Makefiles

You can create PWB makefiles from PWB menus, and NMAKE makefiles with any editor, but the two files are not the same. NMAKE makefiles are fully customizable, PWB makefiles are not. For example, when you create PWB makefiles with the Set Program List command, optimizations set with commands Build Options, C Compiler Options, and Link Options affect all object modules equally. With NMAKE, you can customize optimizations for sets of object modules.

What is more, you cannot edit PWB makefiles like NMAKE makefiles. PWB creates build information that you cannot disturb without corrupting the file. Nevertheless, you can edit PWB makefiles in these ways:

- Use the Editor Settings command to modify build switches in the <Assign> pseudofile.
- Add your own build information to a PWB makefile by adding the following line at the end of the file:

  ```
  # << User_Supplied_Information >>
  ```

  The comment must begin in the first column, and appear exactly as shown.
- Purposely corrupt the PWB makefile, modify any information, and turn it into a foreign makefile. Once you do this, however, you cannot use the Edit Program List command.

You can, however, execute PWB makefiles with NMAKE. In fact, when PWB builds from a PWB makefile, it uses NMAKE to drive the build. To use NMAKE or any foreign makefile from PWB, select [X] Use as a Non-PWB Makefile from the Set Program List dialog box.

NMAKE with Batch

NMAKE executes commands in makefiles two ways: inline and deferred. The inline method requires that you install the Task Management Service. Once a target is evaluated for update, the commands to update the target are executed in a secondary partition; NMAKE remains in memory. The deferred method builds a submit file for later processing. Targets evaluated for update have the updating commands appended to this submit file. Once the targets have been evaluated fully, NMAKE leaves memory, and the processing of the submit file begins.

As long as you install Task Management Service, NMAKE spawns commands in makefiles. However, NMAKE creates submit files for commands in makefiles if TMS is not installed, if you execute NMAKE with /Y, or if you set the environment variable MAKEFLAGS=Y.

This distinction between spawning and submit files becomes an issue with Batch files. NMAKE runs properly from Batch when it spawns commands. NMAKE generally does not run properly from Batch when it builds submit files. Batch specifies itself as the Exit run File and calls Chain to run NMAKE, which builds a submit file and uses Chain to start it.

When the first command of the NMAKE submit file completes, the Exit run file executes, not Batch. Batch continues to execute and the NMAKE submit file does not resume until Batch finishes. Nevertheless, the Batch-NMAKE combination succeeds if NMAKE is run last in the Batch file, or if NMAKE's execution results in a single command.

Programming Notes

This subsection covers these topics:

- Using the _plm Keyword
- Output Redirection
- Input Redirection
- Handling Nameless Structs and Unions
- Manipulating Error Messages
- Binary and Text Mode
- Function Differences
- Functions Not Supported

Using the _plm Keyword

For functions produced by PL/M and the CT Assembler, CTOS Microsoft C 6.0 required you to use Envelope functions, which provided a workaround to protect the SI and DI registers from being corrupted by code found primarily in CTOS.Lib. The 6.0 version of CTOS.Lib.h contained _cdecl prototypes for these Envelope functions. h

With 6.1, CTOS.Lib.h contains _plm prototypes for the CTOS.Lib functions themselves. The 6.1 C compiler generates code to preserve SI and DI registers around calls to _plm functions. In addition, a CLD instruction is emitted after each _plm call to ensure a cleared direction flag.

To use the _plm keyword on new programs:

- Use the 6.1 header file CTOS.Lib.h for CTOS.Lib prototypes.

If you created Envelope function libraries:

- Replace _cdecl with _plm on all prototypes in your header files.
- Recompile, and link without Envelope libraries.

_plm Keyword Limitations

The 6.1 PL/M function call implementation retains two limitations:

- Calls to _plm function pointers are NOT register protected as are calls to _plm functions. (This is also true of BTOS C's plm implementation).
- The return of a far pointer in the ES:BX registers by BTOS C 'plm' functions or PL/M functions is not supported.

Although the incidence of ES:BX far pointer return is extremely rare in existing PL/M libraries, the BTOS C frequency is likely to be higher. If you have the source code for BTOS C 'plm' functions that return a far pointer, simply port the BTOS C code to CTOS Microsoft C.

If you do not have access to the source code containing the functions returning ES:BX, an Envelope function strategy is recommended. Although the Envelope approach requires you to create an Envelope function source and object, no changes are required to your original source code. Simply recompile with the _cdecl Envelope prototype and re-link with the new Envelope object. Compile all required Envelope functions with these suggested options:

```
/AL /G1s /Zl /NTESBX_INTF_TEXT
```

Here is a sample Envelope source:

```
typedef  void _far *  Pvoid ;
void _far _plm PLM_FUNCTION( args ) ; // The function
                                      // returning ES:BX
      // The Envelope function:

   Pvoid _far _cdecl Plm_Function( args )
   {

     PLM_FUNCTION( args ) ; // Call the REAL one

     _asm    mov  dx, es
     _asm    mov  ax, bx

   }
```

And a sample Envelope function usage:

```
typedef  void _far *  Pvoid ;

Pvoid _far _cdecl Plm_Function( int args ) ;

main()
{

  int        i ;
  char _far *p ;

      ...

  p = Plm_Function( i ) ;

      ...

}
```

In this example, the _cdecl Envelope function Plm_Function( ) calls the _plm function PLM_FUNCTION( ) with the identical arguments. The ES:BX registers are moved to DX:AX just prior to the Envelope function's return so that the assignment in main( ) is correct.

Output Redirection

Output redirection is a behavior inherited by all CTOS Microsoft C applications linked with the standard startup objects, C Library, and LIBCEXT.LIB. Since all utilities in this release except QuickHelp fall into this category, you may specify output redirection on any of the command forms except the QH command form.

The syntax for output redirection is:

```
>Output_FileSpec
```

where Output_FileSpec names a file to which all stdio and stderr output is written. If Output_FileSpec does not exist, it is created. No spaces are allowed between the > character and file name. The output redirection argument can appear in any argument position on the command line.

The syntax to specify appending redirected output to a file is:

```
>>Output_FileSpec
```

Output redirection also allows you to independently redirect stdout and stderr.

The syntax to redirect stdout is:

```
1>Output_FileSpec
```

To redirect stderr, use:

```
2>Output_FileSpec
```

Input Redirection

Input redirection of stdin requires the syntax:

```
<Input_FileSpec
```

As with output redirection, no spaces are allowed between the '<' character and the file name. The input redirection argument may appear in any argument position on the command line.

Handling Nameless Structs and Unions

You cannot use a tag or typedef to add the members of an anonymous (nameless) struct or union to another anonymous struct or union. Instead, you must give the full definition of the struct to be nested. For example, assume you have the following anonymous struct:

```
struct oneTag
{
        int aMem;
        int bMem;
};
```

The following code fragment shows the incorrect way to include the members of this struct in another anonymous struct:

```
struct anotherTag
{
        struct oneTag;   //  INCORRECT
        int cMem;
};
```

The correct method is to fully specify the nested struct:

```
struct anotherTag
{
        struct           //  CORRECT
        {
                int aMem;
                int bMem;
        };
        cMem;
};
```

Manipulating Error Messages

To print error information, use the perror function. When an error occurs in a system-level call, the errno variable is set to an integer value to reflect the type of error.

The perror function uses the errno value to look up the corresponding error message in the sys_errlist table. The value of the sys_nerr variable is defined as the number of elements in the sys_errlist array.

The value of errno reflects the error value for the last call that set errno. The errno value is not automatically cleared by later successful calls. Thus, to obtain accurate results, immediately after a call you should test for errors and if desired, print error messages.

To access the actual CTOS error code, use the _ctoserrno variable. When an error occurs in a CTOS call, the _ctoserrno variable holds the actual error most recently returned.

Variable: _ctoserrno, errno, sys_errlist, sys_nerr

Include: <stdlib.h>, <errno.h>

Syntax:
```
unsigned short _ctoserrno;
int errno;
char * sys_errlist[ ];
int sys_nerr;
```

The following errno values are supported:

E2BIG	ECHILD	EINVAL	ENOMEM
EACCES	EDEADLOCK	EMFILE	ENOSPC
EAGAIN	EDOM	ENOENT	ERANGE
EBADF	EEXIST	ENOEXEC	EXDEV

Binary and Text Modes

The default open file mode is binary (untranslated, O_BINARY), not text mode (O_TEXT). To force a text mode default, link the file TxtMode.Obj with your program. As an alternative, remove BinMode.Obj from the C Library files and replace it with TxtMode.Obj.

The default line ending for CTOS text files is a single linefeed character (LF, 0x0a), as opposed to the carriage-return linefeed combination (CRLF, 0x0d0a). Your programs will output a CRLF line ending only if you explicitly code the \r\n escape characters in an output specification.

Binary mode provides completely untranslated I/O. Text mode differs from binary in two ways:

- CRLF combinations are translated to a single LF on input
- The CTRL+Z (^Z) character is interpreted as an end-of-file on input.

Miscellaneous Function Behaviors

The system function returns 0 if a NULL argument is supplied; otherwise, it returns 1. No command processor is spawned; no command is executed.

The _makepath, _splitpath, and _fullpath functions use CTOS path semantics.

The exec function group has been implemented using the CTOS.LIB Chain function. These exec functions do not support the use of CTOS subparameters.

The stat function for getting status information on a file cannot be used for an open file because CTOS does not allow a file to be opened more than once. Use the fstat function for getting status information on an open file.

Functions Not Supported

Some obvious functional areas of the original Microsoft C Library cannot be supported, most notably those functions that are found in the original ONLY in the real mode libraries (xLIBCR). These unsupported groups include the BIOS and DOS interface groups _bios* and _dos*. Table 5-11 lists other functions in this category.

Table 5-11. Unsupported Library Functions

Unsupported Category	Function
DOS BIOS & Interface	All _bios*
	All _dos*
	bdos
	dosexterr
	int*
	_chain_intr
	_disable
	_enable
	_hard*
Drive number usage	_chdrive
	_getdrive
	_getdcwd
File locking	_fsopen
	locking
File sharing	sopen

continued

Table 5-11. Unsupported Library Functions (cont.)

Unsupported Category	Function
Pipes	_pclose
	_pipe
	_popen
Child process management	spawn*
	cwait
	wait
Signal management	abort
	raise
	signal

Section 6
Optimizing C Programs

The CTOS Microsoft C compiler translates C source statements into machine-executable instructions. By default, the compiler rewrites or optimizes your source code to make it more efficient. In addition, you can specify other goal-oriented optimizations when compiling and linking. For instance, you can tell the compiler to produce the fastest code possible, and tell the linker to pack adjacent code segments.

To explain optimization, and show how to control it, this section is divided into the following topics:

- How to Specify Compiler Optimization
- Default Optimization
- Customizing Optimization
- CTOS Linker Optimization
- Function-Calling Conventions

How to Specify Compiler Optimization

By default, the compiler optimizes code in three general ways:

- Modifies or moves sections of code to use fewer instructions and use the processor more efficiently.
- Moves code and combines operations to maximize register use, which results in faster code.
- Eliminates sections of redundant or unused code.

You can specify additional compiler optimizations from three places:

- Programmer's WorkBench
- Command Line
- Source Code with Pragmas

Optimizing from Programmer's WorkBench (PWB) and from the command line produces the same effect: global optimization for all compiled source code. To select options from PWB, you use a mouse and pull down menus. To select options from the CL command line, you type switches such as /Os, which generates the smallest possible code.

Optimizing with pragmas in source code allows you to apply different optimization controls to individual functions within a single source module.

Optimizing from Programmer's WorkBench

The Programmer's WorkBench (PWB) is an integrated development environment used for editing, building, and debugging applications written in Microsoft C.

PWB allows you to choose two types of compiles: debug and release. Without user modification, the two types default to the lower and upper limits of optimization:

- Debug compile.

  Without user modification, the compiler performs no optimization.

- Release compile.

  Without user modification, the compiler performs most optimizations.

By modifying settings for Debug Build and Release Build in the Compiler Options menu, you can fine tune both compiles. For the most part, PWB menu options correspond to CL command-line options. In fact, the PWB constructs a command line from your input and passes it to CL.

Note: *For convenience, this section describes each optimization option in terms of command-line switches and pragmas. You can also select all options from the PWB Compiler Options menu.*

Optimizing from the Command Line

To control optimization from the command line, first decide what optimizations your application needs. Then specify these optimizations using command-line option switches. Optimization switches generally begin with /O, but some begin with /G.

The command line:

```
CL
  [Args]     /Oa /Ol /Ot TEST.c
```

compiles the program TEST.c, and instructs the compiler to:

- (/Oa) Optimize without aliasing
- (/Ol) Optimize loops
- (/Ot) Optimize for speed

The previous command line can also appear as:

```
CL
  [Args]     /Oalt TEST.c
```

Optimizing from Source Code with Pragmas

Pragmas, special lines of code that start with #pragma, allow finer control over optimization than command-line options. With pragmas, you can optimize individual functions rather than an entire source file.

CTOS Microsoft C supports several pragmas that provide control of the following optimizations:

- t Generate faster code (on) or smaller code (off)
- a Assume no aliasing
- w Assume no aliasing, but reload pointer variables
- l Optimize loops
- n Disable unsafe loop optimizations
- z Optimize aggressively

- e Allocate global registers
- c Reduce local common subexpressions
- g Reduce global common subexpressions
- p Achieve consistent floating-point results
- u Remove SI and DI preserving code
- y Remove stack frame

Using the optimize pragma, you can enable or disable any optimization or combination of options. For example, if you have one function that uses aliases heavily, you need to inhibit optimizations that could cause problems with aliases. You do not, however, want to inhibit these optimizations for code that does not do aliasing. To do this, use the optimize pragma as follows:

```
/* Function(s) that do not do aliasing. */
.
.
.
#pragma optimize("a", off)
/* Function(s) that do aliasing. */
.
.
.
#pragma optimize("a", on)
/* More function(s) that do not do aliasing. */
```

To enable or disable multiple options, combine pragma parameters in a string.

For example,

```
#pragma optimize("lge", off)
```

disables loop optimization, global common subexpression optimization, and global register allocation.

Default Optimizations

Almost all applications benefit from the default compiler optimizations. These optimizations are 0small in scope and are almost always helpful. They include:

- Common subexpression elimination
- Dead-store elimination
- Constant propagation

Nevertheless, sometimes you may want to disable default optimization, which may rearrange code in the object file, making it difficult to recognize when debugging, especially with a symbolic debugger. To remove all optimization, use option /Od (disable optimizations).

To disable all optimizations for a function, use this pragma:

```
#pragma optimize("", off)
```

To restore optimization, use this pragma:

```
#pragma optimize("", on)
```

Common Subexpression Elimination

In common subexpression elimination, the compiler finds code that contains repeated subexpressions, and modifies it to evaluate subexpressions once only. Subexpression elimination is usually accomplished with temporary variables as shown in this example:

```
a = b + c * d;
x = c * d / y;
```

These lines contain the common subexpression c * d. This code can be modified to evaluate c * d once only. The result appears in a temporary variable (usually a register):

```
tmp = c * d;
a = b + tmp;
x = tmp / y;
```

Dead-Store Elimination

Dead-store elimination is an extension of common subexpression elimination. With this option, the compiler combines variables that contain identical values into one temporary variable.

In the following code fragment, the compiler detects that the expression func(x) is equivalent to func(a + b):

```
x = a + b;
x = func(x);
```

Thus, the compiler rewrites the code as:

```
x = func(a + b);
```

Constant Propagation

When optimizing constant propagation, the compiler analyzes variable assignments and decides if they can be changed to constant assignments. In the following example, variable i must have a value of 7 when assigned to j:

```
i = 7;
j = i;
```

However, instead of assigning i to j, the compiler can assign the constant 7 to j:

```
i = 7;
j = 7;
```

While you could change these variables in the source file, doing so might reduce readability. Thus, in many cases, optimizations allow you to write easier-to-read code without losing efficiency.

Customizing Optimizations

While default optimizations are sufficient for many applications, you may want to tune your programs according to criteria not known to the compiler. The optimization options are:

- Fast Speed (/Ot)
- Small Size (/Os)
- Generate Instrinsic Functions (/Oi)
- Assume No Aliasing (/Oa)
- Assume No Aliasing, but Reload Pointer Variables (/Ow)
- Optimize Loops (/Ol)
- Disable Unsafe Loop Optimization (/On)
- Allow Aggressive Optimization (/Oz)
- Remove Stack Probes (/Gs)
- Enable Global Register Allocation (/Oe)
- Enable Global Common Subexpressions (/Og)
- Enable Default Common Subexpression (/Oc)
- Achieve Consistent Floating-Point Results (/Op)
- Generate Code for Processors 8088 to 80486 (/G0)
- Generate Code for Processors 80188 to 80486 (/G1)
- Generate Code for Processors 80286 to 80486 (/G2)
- Optimize for Maximum Efficiency (/Ox)
- Remove SI and DI Preserving Code (/Ou)
- Remove Stack Frame Generation Code (/Oy)

The default optimizations are /Oc, /On, and /Ot.

Choosing Speed or Size (/Ot and /Os)

The CTOS Microsoft C compiler automatically uses option /Ot, which optimizes for speed. However, while option /Ot increases speed, it may also increase size.

If you would rather optimize for small program size, use option /Os. Option /Os decreases program size, but may also decrease program speed.

To optimize for speed or size on a per-function basis, use the optimize pragma with option t. The on setting instructs the compiler to optimize for speed; the off setting instructs the compiler to optimize for compactness of code.

Here are examples:

```
#pragma optimize("t", off)     /* Optimize for smallest code. */
.
.
.
#pragma optimize("t", on)     /* Optimize for fastest code. */
```

Generating Intrinsic Functions (/Oi)

In place of some normal function calls, the C compiler can insert intrinsic functions, which operate faster. Every time a function is called, a set of instructions is executed to store parameters and to create space for local variables. When the function returns, more code is executed to release space used by local variables and parameters and to return values to the calling routine. These instructions take time to execute. In the context of an average-sized function, the additional code is minimal, but if the function is only a line or two, the additional code can comprise almost half of the function's compiled code.

One way to avoid this type of code expansion is to avoid such short functions, especially in often-used sections of code where speed is critical. But many library functions contain only a line or two of code. The compiler provides two forms of certain library functions. One form is a standard C function, which requires the overhead of a function call. The other form is a set of instructions that performs the same action as the function without issuing a function call. This second form is called an intrinsic function. Intrinsic functions are always faster than their function-call equivalents and can provide significant optimizations at the object-code level.

For example, the function strcpy might be written as follows:

```
int strcpy(char * dest, char * source)
{
  while(*dest++ = *source++);
}
```

The compiler contains an intrinsic form of strcpy. If you instruct the compiler to generate intrinsic functions, the compiler replaces calls to strcpy with the intrinsic form.

Note: *While the example above is written in C for clarity, most library functions use assembly language to take full advantage of the 80x86 instruction set. Intrinsic functions are not simply library functions defined as macros.*

Compiling with the /Oi option causes the compiler to use the intrinsic forms of the following functions:

abs	labs	outp	strcpy
_disable	lrotl	outpw	strlen
_enable	lrotr	rotl	strset
fabs	memcmp	rotr	
inp	memcpy	strcat	
inpw	memset	strcmp	

The following floating-point functions do not have true intrinsic forms, but do have versions that pass arguments directly to the floating-point chip instead of pushing them onto the normal argument stack:

acos	fmod	acosl	fmodl
asin	log	asinl	logl
atan	log10	atanl	log10l
atan2	pow	atan2l	powl
ceil	sin	ceill	sinl
cos	sinh	cosl	sinhl
cosh	sqrt	coshl	sqrtl
exp	tan	expl	tanl
floor	tanh	floorl	tanhl

Note: *The compiler performs optimizations assuming math intrinsics have no side effects. This assumption holds true except for user-written matherr functions that alter global variables. If you write a matherr function to handle floating-point errors and your function has side effects, use the function pragma to instruct the compiler not to generate intrinsic code for math functions.*

To generate intrinsic functions for a subset of the functions listed above only, use the intrinsic pragma rather than the /Oi option. The intrinsic pragma has the following format:

```
#pragma intrinsic(function1, ...)
```

The following code illustrates how to use the intrinsic pragma:

```
#pragma intrinsic(abs)
void main(void)
{
  int i, j;
  i = big_routine_1();
  j = abs(i);
  big_routine_2(j);
}
```

In this example, an intrinsic function replaces the abs call with assembly-language code that takes the absolute value of a number. Thus, the program executes faster because the function-calling overhead is no longer required. However, because abs is called once only, the overall speed increase is small.

In the next example, the call to abs is in a loop. So in this instance, you can save a significant amount of execution time by generating intrinsic functions.

```
#pragma intrinsic(abs)
void main(void)
{
  int i, j, x;
  for(j = 0; j << 1000; j++)
  {
    for(i = 0; i << 1000; i++)
    {
      x += abs(i - j);
    }
  }
  printf("The value of x is %d\n", x);
}
```

Two restrictions exist for using intrinsic forms of function calls:

- Do not use intrinsic forms of the floating-point math functions with alternate math libraries (xLIBFA.LIB).
- If you do not want to generate instrinsic functions, be careful when using option /Ox (maximum optimization), which enables option /Oi (generate intrinsic functions). You may have to use pragmas.

Assuming No Aliasing (/Oa and /Ow)

An alias is a name that refers to a memory location already referred to by another name. Because accessing memory takes longer than accessing CPU registers, the compiler tries to store frequently-used variables in registers. However, aliasing reduces the extent to which a compiler can keep variables in registers.

A pointer refers to a memory location. Because pointer values are not known until a program runs, the compiler does not know which memory location the program will modify. Therefore, the compiler must assume that any time a pointer value changes, the value of other variables might also change. This limits the extent to which the compiler can move values from memory to registers.

Option /Oa tells the compiler to ignore the possibility of multiple aliases for a memory location. In the list that follows, the term reference means read or write: whether a variable on either the left or right side of an assignment statement is a reference. In addition, any function calls that use a variable as a parameter are references to that variable. When you tell the compiler to assume that you are not doing aliasing, it expects that the following rules are being followed for any variable not declared as volatile:

- If a variable is used directly, no pointers are used to reference that variable.
- If a pointer is used to refer to a variable, that variable is not referred to directly.
- If a pointer is used to modify a memory location, no other pointers are used to access the same memory location.

To clarify how these rules affect your code, consider the following example:

```
char     p;
char     *ptr_p;
ptr_p = &p;       /* Take the address of p. */
```

You can now refer either to *ptr_p or to p, but not to both within the same function. If you must refer to the variable by both names, you are using aliases.

Code referring to the same location with two pointers uses aliases. For example:

```
char     *p_buf;
char     *p_alias;
if((p_alias = p_buf = malloc(5000)) == NULL)
 return;
else
{
   .
   .
   .
}
```

The code illustrated in the previous example is common. It demonstrates dynamic allocation of a block of memory from the heap, preserving the original address in p_buf. The program then performs all pointer arithmetic on the alias p_alias. When the function finishes with the block of memory, p_buf is a valid argument for the free function, because it still contains the original address.

The difference between options /Oa and /Ow, is that /Oa specifies that you will not be doing aliasing (which allows the compiler to perform significant optimizations that might not otherwise have been possible), and that function calls are safe. Option /Ow is similar to /Oa, except that after a function call, pointer variables must be reloaded from memory.

Here is an example that makes a poor candidate for both /Oa and /Ow optimization:

```
int g;
void main(void)
{
  add_em(&g);
}

int add_em(int *p)
{
  *p = 2;       /* Assign a value to an alias for g.  */
  g = 3;        /* Assign a value directly to g.  */
  return(*p + g);
}
```

In the function add_em, both g and *p refer to the same memory location. This location is first assigned 2, then 3. The value pointed to by *p (the alias for g) is then added to g, and the result is returned to the main program. If you do not use the /Oa command-line option, the compiler assumes that the reference to *p could refer to the same memory location as does g, and makes no attempt to use a register to store the value of either one. If, however, you do specify option /Oa, the compiler assumes that g and *p refer to different memory locations, and stores them in different registers. At the return statement, g will have a different value than *p, even though both aliases should actually contain the same value.

Note that the compiler keeps values in registers for a limited time only. If different aliases to a memory location occur in different functions, they will not cause unexpected results. When in doubt, avoid aliasing.

Aliasing bugs most frequently show up as corrupted data. If you have global or local variables being assigned seemingly random values, take the following steps to determine if you have a problem with optimization and aliasing:

- Compile the program with /Od (disable optimizations).
- If the program works when compiled with option /Od, check your normal compile options for option /Oa (assume no aliasing).
- If you were using option /Oa option, fix your compile options so /Oa is not specified.

Note: You can use the optimize pragma with option a or w to instruct the compiler to disable optimizations that are unsafe with code that does aliasing.

Performing Loop Optimizations (/Ol)

Option /Ol enables a set of optimizations involving loops. Because loops involve sections of code that are executed repeatedly, they are targets for optimization. These optimizations all involve moving or rewriting code so it executes faster.

Loop optimization can be turned on with option /Ol, or with pragma loop_opt. The following line enables loop optimization for all subsequent functions:

```
#pragma loop_opt(on)
```

The following line turns it off:

```
#pragma loop_opt(off)
```

An optimal loop contains only expressions that change values with each execution. Any subexpression that contains constant values should be evaluated before the body of the loop is executed. Unfortunately, these subexpressions are not always readily apparent. The optimizer can remove many of these expressions from the body of a loop at compile time. This example illustrates invariant code in a loop:

```
i = -100;
while(i < 0)
{
  i += x + y;
}
```

In this example, the expression x + y does not change in the loop body. Loop optimization removes this subexpression from the body of the loop so that it executes once only, and not during every loop execution. The optimizer changes the code to the following fragment:

```
i = -100;
t = x + y;
while(i < 0)
{
  i += t;
}
```

Loop optimization is much more effective when the compiler can assume no aliasing. While you can use loop optimization without option /Oa or /Ow, use /Oa to ensure that the compiler uses the most options possible.

Here is a code fragment that could have an aliasing problem:

```
i = -100;
while(i < 0)
{
  i += x + y;
  *p = i;
}
```

If you do not specify option /Oa, the compiler must assume that either x or y could be modified by the assignment to *p. Therefore, the compiler cannot assume that the subexpression x + y is constant for each loop iteration. If you specify that you are not doing any aliasing (with option /Oa), the compiler assumes that modifying *p cannot affect either x or y, and that the subexpression is indeed constant and can be removed from the loop, as in the previous example.

Note: *All loop optimizations specified by the option /Ol or pragma loop_opt are safe optimizations. To enable aggressive loop optimizations, you must use the enable aggressive optimization, option /Oz. While the optimizations enabled by the combination of /Ol and /Oz are not safe for all cases, they work properly for most programs.*

Disabling Unsafe Loop Optimizations (/On)

The disable unsafe loop optimizations (/On) option is obsolete, and is retained for compatibility with existing makefiles only. Loop optimizations are, by default, safe optimizations. Option /On is the default and has the opposite effect of option /Oz (enable aggressive optimizations).

Enabling Aggressive Optimizations (/Oz)

The compiler can perform extremely aggressive optimizations. These optimizations produce high code quality both in terms of speed and size. Certain programs, however, cannot be optimized with the technologies enabled by the /Oz option. For these programs, you should not specify this option. You can, however, use all other optimization options.

Because the optimization strategies enabled by option /Oz are so aggressive, they are not part of the maximum optimization (/Ox) option.

Option /Oz produces the following effects:

- Loop optimization (/Ol)

  Loop optimization enables a technology that anticipates program flow and tries to remove invariant expressions from loops. When you specify option /Oz, the compiler removes invariant expressions even if errors might occur. In fact, errors with option Oz occur most often when an invariant expression that can cause an exception is protected by an if statement. The invariant expression is hoisted out of the loop body, causing it to be evaluated prior to the evaluation of the if statement that was designed to protect it.

  The following two examples illustrate this problem:

```
for(i = 0; i < 100; ++i)
  if(float_val != 0.0F)
    /* Protect against divide-by-zero.  */
       float_result = pi / float_val;

while(condition)
  if(ptr_val != NULL)
    /* Protect pointer dereference.  */

       char_var = *ptr_val;
```

- Global register allocation (/Oe)

  Option /Oz enables some register allocation strategies that can cause the compiler to place invalid segment selectors in registers, which can cause protection faults in CTOS.

Note: *You can use the optimize pragma with the z option to instruct the compiler to enable aggressive optimizations on a function-by-function basis.*

Removing Stack Probes (/Gs)

Every time a function is called, the stack provides space for all parameters and local variables declared in that function. A short assembly function in then called to check for a stack overflow. Stack overflows are usually caused either by infinite loops, or by runaway recursive routines. Such errors can also be caused by extremely large parameters or local variables.

Therefore, stack probes can be important during program development, because stack-overflow errors alert you to problems in your code. After testing, however, stack checking often becomes unnecessary. The compiler allows you to remove stack-checking code with either option /Gs, or with pragma check_stack. Eliminating stack probes produces smaller and faster programs.

Enabling Global Register Allocation (/Oe)

The global register allocation option (/Oe) instructs the compiler to analyze your program and allocate CPU registers as efficiently as possible. Without the global register allocation option, the compiler uses the CPU's registers for several purposes:

- Holding temporary copies of variables
- Holding variables declared with the register keyword
- Passing parameters to functions declared with the _fastcall keyword (or functions in programs compiled with the /Gr command-line option)

When you enable global register allocation, the compiler ignores the register keyword and allocates register storage to variables (and possibly to common subexpressions). The compiler allocates register storage to variables or subexpressions according to frequency of use. Because of the limited number of physical registers, variables held in registers are sometimes placed back in memory to free the register for another use.

The following C program example demonstrates how the compiler might rewrite your code to accomplish this:

```
/* Original program */

func()
{
     int i, j;
     char *pc;

     for(i = 0; i << 1000; ++i)
     {
         j = i / 3;
         *pc++ = (char)i;

     }

     for(j = 0, --pc; j << 1000;
          ++j, --pc)
          *pc--;
}

/* Example of how the compiler might optimize the
* code to move i and j in and out of registers */

func()
{

     int i, j;
     char *pc;
     {
     register int i; /* i is in a register for this block.  */
          for(i = 0; i << 1000; ++i)
          {
               j = i / 3;
               *pc++ = (char)i;
          }

     }

     {
     register int j; /* j is in a register for this block.  */
          for(j = 0, --pc; j << 1000;
               ++j, --pc)
               *pc--;

     }
}
```

In the preceding example, notice the blocks enclosed in curly braces used to delimit the span of code across which variables should remain in registers.

Note: *To enable or disable global register allocation on a function-by-function basis, use the optimize pragma with option e. The /Oe optimization is incompatible with /r, suppress register variables.*

Enabling Common Subexpression Optimization (/Oc and /Og)

When using option /Og (enable global common subexpression optimizations), the compiler searches entire functions for common subexpressions. Option /Oc (default common subexpression optimization) examines only short sections of code for common subexpressions. You can disable default common subexpression optimization with the /Od option. For more information about common subexpression optimization, refer to the heading Default Optimization.

Note: *To enable or disable block-scope common subexpression optimization on a function-by-function basis, use the optimize pragma with option c . To enable or disable global common subexpression optimization on a function-by-function basis, use the optimize pragma with option g. /r disables /Oc; /r is incompatible with /Og.*

Achieving Consistent Floating-Point Results (/Op)

Floating-point numbers stored in memory use either 32, 64, or 80 bits, depending on whether they are of type float, type double, or type long double. The 80x87 family of coprocessors uses 80-bit registers for all operations. If a value of type float or type double is kept in these registers through a number of operations, it retains more accuracy than when moving that value to and from memory between operations.

Because of the difference in precision between memory and register representation of a floating-point number, a value stored in memory is not always equal to the same value in the 80x87 register.

The difference in precision primarily affects strict equality or strict inequality tests (== and !=); however, relational tests of magnitude (>, >=, <=, and <) can behave erroneously if the coprocessor is able to maintain significant digits that memory variables cannot.

To avoid the precision difference, use option /Op. This option forces floating-point values to be written to memory between floating-point operations. While storing these values to memory reduces the precision of floating-point expressions, it does ensure that these expressions will produce consistent results.

To change the handling of floating-point results on a function-by-function basis, use the optimize pragma with the option p.

Note: *Option /Op suppresses other optimizations because the floating-point registers are not available for storage of intermediate results. Because these optimizations are suppressed, code compiled with option /Op executes slower than code compiled without this option. Careful coding practices, especially in tests of strict equality and inequality, can alleviate the need for this option.*

Using the 80186, 80188, or 80286 Processor (/G0, /G1, /G2)

By default, the compiler generates 8086 object code (/G0). Because the newer processors (the 80186, 80188, 80286, 80386, and 80486) are backward-compatible with the 8086 instruction set, using this instruction set ensures compatibility for all 80x86-based computers. However, while you gain compatibility, you lose the advantage of some powerful instructions in the newer processors.

If you know your program will be running on an 80186 or 80286 processor only, you can cause the compiler to generate instructions specific to these processors. These instructions increase your program speed, but you lose compatibility with machines that use older processors in the 80x86 family. Table 6-1 lists the options for processor-specific code generation:

Table 6-1. Processor Compatibility

Command-Line Option	Compatible Processors
/G0	8088, 8086, 80186, 80286, 80386, 80486
/G1	80186, 80286, 80386, 80486
/G2	80286, 80386, 80486

Note: When developing exclusively for protected mode BTOS or CTOS, always use option /G2, because these operating systems do not run on the 8086, 8088, or 80186.

Optimizing for Maximum Efficiency (/Ox)

Option /Ox combines a number of different optimizations:

- Enable global register allocation (/Oe)
- Enable global common subexpression optimization (/Og)
- Enable block-scoped common subexpression optimization (/Oc)
- Generate intrinsic functions (/Oi)
- Perform loop optimizations (/Ol)
- Optimize for speed (/Ot)
- Remove stack probes (/Gs)

Option /Ox does not include several optimizations that can improve code efficiency: /Oa (assume no aliasing), /Oz (enable aggressive optimizations), and /Gr (use fastcall calling convention). To determine if these optimizations are appropriate for your applications, , you should read the text describing options /Oa and /Oz, and the text describing the fastcall calling convention.

If you are more concerned with executable file size than with execution time, use options /Ox and /Gs, then issue the optimize pragma as follows:

```
#pragma optimize("t", off)
```

This set of options produces the smallest possible code, while also performing some speed optimizations.

Function Prolog and Epilog Optimization (/Ou and /Oy)

Microsoft C 6.1 changes the default code generation for function prologs and epilogs in two ways:

In 6.0, the compiler-generated prolog and epilog code that preserved the SI and DI registers was removed from functions that did not use these registers, unless optimization was disabled with option /Od. For 6.1, the register-preserving code is not removed unless option /Ou is specified, and the DI and SI registers are not used in the function. In other words, /Ou specifies do not save and restore SI and DI registers at the function prolog and epilog if the function does not use them. The associated pragma is:

```
#pragma optimize ( "u", on|off ).
```

You should not use /Ou with functions that call _plm functions, and are not themselves _plm functions because SI and DI need to be protected on behalf of the caller. For example:

```
c_func_01()
{
   ...          // Possible SI and DI usage
   c_func_02() ;
   ...          // SI/DI corrupted by plm_func
}

c_func_02()  // Does not use SI and DI so /Ou
{            // will remove prolog/epilog code:
   ...       // SI/DI protection optimized out!
   ...
   plm_func() ;
   ...
   ...       // SI/DI may now be incorrect for
             // c_func_01
}
```

In this case, c_func_02() must retain the prolog/epilog protection for SI and DI even though c_func_02() does not use SI and DI. If c_func_02() was a _plm function, the /Ou optimization could be safely applied.

Similarly, the 6.0 compiler always generated code to establish a stack frame for each function, but removed that code for functions without arguments and local variables unless all optimization was disabled using option /Od. For 6.1, however, the stack frame generation code is not removed for those eligible functions unless the /Oy option is specified. This change provides, by default, code that conforms to the function entry requirements of the CTOS Virtual Code Segment Management Facility for programs designed to use overlays. The associated pragma is:

```
#pragama optimize ( "y", on|off ).
```

CTOS Linker Optimizations

The CTOS Linker provides two ways to speed program execution: by translating far calls to near calls, and by packing adjacent code segments. The CTOS Linker reads parameter data from a configuration file, where you specify these optimizations. You can also specify these optimizations using MLINK command-line options.

The text that follows covers these topics:

- Linker Configuration File
- Enabling Far Call Optimization
- Packing Code

Linker Configuration File

The Linker configuration file is a text file named LinkerConfig.sys that you may use to specify parameters or library references. The entries in the configuration file have the standard CTOS configuration file format:

```
:tag:data
```

where *tag* is the option name and data is a value such as Yes or No, which toggles the option on or off, or a number.

You can set most existing and all new options using the configuration file. If the Linker cannot find any configuration file or if no entry is made in the configuration file for a parameter, the Linker will use the default in the Linker code.

MLINK options support most config file parameters. For a cross-referenced list of MLINK options and config file parameters, refer to Section 5, Building Applications.

Enabling Far Call Optimization (FarCallTranslation)

Functions are called two ways. In a near call, both the calling statement and the function must be located in the same segment. Only the offset is used to access the function; the segment address is implicit. In a far call, the function is called using both the segment and the offset of the function. This allows a call a routine in another segment.

Because of processor architecture, near function calls execute faster than far calls. The decision to declare functions as near or far is often made when selecting a memory model. As it is difficult to determine where the linker will place a given function in memory, it is impractical for the programmer to choose the way a function is called.

The FarCallTranslation option enables far call optimization. When you use this option, any function calls within the same segment as the function being called are converted to near calls. This optimization has no effect if you compiled for the small or compact model, because all calls are already near calls.

Linker configuration file syntax for the far call translation option is:

```
:FarCallTranslation:Yes
```

The MLINK option to enable far call translation is /FAR. The default is off.

How FarCallTranslation Affects Your Code

FarCallTranslation allows functions to be called with near calls in all possible instances. To perform this translation, the linker takes a section of object code such as:

```
CALL        FAR         _func
```

where func is defined in the current segment, and replaces it with the following code:

```
PUSH        CS
CALL        NEAR        _func
```

This substitution works because the linker has inserted PUSH CS to place a far return address on the stack.

The FarCallTranslation option is most effective when used in conjunction with the PackCode option (MLINK option /PACKC). Using the PackCode option causes far calls to another segment to become calls within the same segment. The FarCallTranslation feature can then take advantage of the new grouping to translate far calls into near calls.

Note: *In some cases, especially if the object module is not generated by a high-level language, the Linker may inadvertently overwrite an instruction sequence that it assumes starts with a far call, but in fact does not start with a far call.*

Benefits of FarCallTranslation

Using FarCallTranslation results in faster code. Since near calls are not recorded in the relocation tables of the run file, run files will also be slightly smaller.

The FarCallTranslation option is of significant benefit to protected-mode programs, as Table 6-2 illustrates.

Table 6-2. Processor Clock Cycles for Calling Sequence

	Protected Mode Cycles	
Instruction	**286**	**386**
Far Function Call		
CALL FAR PTR _func	26	34
Total	26	34
Near Function Call		
PUSH CS	3	2
CALL NEAR PTR _func	7	7
NOP	3	3
Total	13	12
Savings	13	22

Packing Code (PackCode)

The PackCode linker option groups neighboring code segments together. When used with the FarCallTranslation option, the PackCode option greatly increases the number of near calls that can be made to a function. This option can be followed with a segment-size limit (expressed in bytes) at which to stop packing and to begin a new segment. The Linker configuration file syntax for PackCode is:

;PackCode:*data*

where *data* is one of the choices in the following table.

***Data* Value**	**Meaning**
No	Adjacent code segments are not combined into the same physical segment. This the default.
Yes	Adjacent code segments are combined into the same physical segment up to 65,536 bytes.
<*number*>	Adjacent code segments are combined into the same physical segment up to the *number* of bytes specified by *number*. The value of number may not exceed 65,536 bytes.

Choosing Function-Calling Conventions

CTOS Microsoft C supports three function calling conventions:

- (/Gd) C
- (/Gc) PLM/Pascal
- (/Gr) Register _fastcall

Note that, while no calling convention has been defined as standard, most C compilers use conventions similar to those described here. The C calling convention requires the most object code to set up, but it is the only calling convention that supports functions with variable-length argument lists. The PLM/Pascal calling convention is more compact, but does not allow for variable-length argument lists. The _fastcall, or register calling convention, is the fastest of the calling conventions, but it does not support variable-length argument lists or mixed-language program interfaces.

The C Calling Convention (/Gd)

Because C allows functions to have a variable number of parameters, parameters must be pushed onto the stack from right to left. (When parameters are pushed from left to right, the location of the leftmost argument on the stack is unknown.) If you do not specify command-line options that modify the function-calling convention, the C calling convention is used; otherwise, the _cdecl keyword must be used before any function using the C calling convention.

If, for example, you use the /Gr (register calling convention) option when you compile, and the function add_two must have the C calling convention, declare add_two as follows:

```
int _cdecl add_two(int x, int y);
```

The PLM/Pascal Calling Convention (/Gc)

Use the PLM/Pascal calling convention for any functions declared with either the _plm or _pascal keywords. (The two keywords currently produce identical results.) Parameters to these functions are always pushed on the stack from left to right. While any function can be declared with the PLM/Pascal convention, it is used primarily for prototypes to Pascal or PLM routines called from within C programs. This calling convention can also produce smaller, faster programs.

To make all functions in a file observe the PLM/Pascal calling convention, use option /Gc (generate Pascal-style function calls).

Note that C run-time library routines must still be called using C calling conventions. Therefore, you must include the appropriate header files in any program using run-time library routines, since the header files contain_cdecl declarations for these routines.

Functions with variable-length parameter lists (such as printf) cannot use the PLM/Pascal calling convention.

The Register Calling Convention (/Gr)

You can decrease execution time if parameters to functions are passed in registers rather than on the stack. Compiling with option /Gr enables the register calling convention for an entire file. The _fastcall keyword enables the register calling convention on a function-by-function basis.

Because the 80x86 processor has a limited number of registers, only the first three parameters are allocated to registers; the rest are passed using the PLM/Pascal calling convention. The register calling convention can increase the speed of a program.

Note: *The compiler allocates different registers for variables declared as register, and for passing arguments using the register calling convention. This calling convention will not conflict with any register variables that you may have declared.*

Exercise caution when using the register calling convention for any function written in in-line assembly language. Your use of registers in assembly-language could conflict with the compiler's use of registers for storing parameters.

Argument-Passing Convention

The _fastcall calling convention is a strongly typed register calling convention. This typing allows the compiler to generate better code by passing arguments in registers that correspond to the data type you are passing. Because the compiler chooses registers depending on the type of the argument and not in a strict linear order, the calling program and called function must agree on the types of the arguments in order to communicate data correctly.

Register candidates exist for each type of argument. The arguments are allocated to registers or, if no suitable register remains unused, are pushed onto the stack left-to-right. Each argument is put in the first register candidate that does not already contain an argument. Table 6-3 shows the basic types and the register candidate list for each.

Table 6-3. Register Candidates

Type	Register Candidates
Character	AL, DL, BL
Unsigned character	AL, DL, BL
Integer	AX, DX, BX
Unsigned integer	AX, DX, BX
Long integer	DX:AX
Unsigned long integer	DX:AX
Near pointer	BX, AX, DX
Far or huge pointer	Passed on the stack

All far and huge pointers are pushed on the stack, as are all structures, unions, and floating-point types.

Return Value Convention

The _fastcall return value convention is based on the size of the return value, except with floating-point types. All floating point types are returned on the top of the NDP stack. The following table shows how values 4 bytes or smaller, including unions and structures, are returned from a _fastcall function.

Size	Return Convention
1 byte	AL Register
2 bytes	AX Register
4 bytes	DX, AX Registers (for pointers, the segment is returned in DX, the offset in AX; for long integers, the most-significant byte is returned in DX, least-significant byte in AX)

Note that the protocol for returning values 4 bytes or smaller is the same as for functions declared as _cdecl. To return structures and unions larger than 4 bytes, the calling program passes a hidden parameter as the last item pushed. This parameter is a near pointer, implicitly SS-relative, to a buffer in which the value is to be returned. A far pointer to SS:hidden-param must be returned in DX:AX. This is the same convention for returning structures as _pascal.

Stack Adjustment Convention

Unlike functions declared as _cdecl, functions declared as _fastcall must pop the arguments off the stack. The calling program does not adjust the stack after a _fastcall function return.

Register Preservation Requirement

All functions must preserve the DS, BP, SI, and DI registers. Your _fastcall function can modify the values in AX, BX, CX, DX, and ES. If your _fastcall function sets the direction flag, it must be cleared before returning.

Function-Naming Convention

The public name put into the object file for a function declared as _fastcall is the name given by the user with a leading at sign (@). No case translation is performed on the function name. The function declaration:

```
int _fastcall FCFunc(void);
```

causes the compiler to place the public symbol @FCFunc in your object file.

If you do not declare the function as _fastcall in your C program, the compiler assumes the default calling convention. The default is the C calling convention but can be changed by the /Gc (PLM/Pascal Calling Convention) or /Gr (_fastcall Calling Convention). If the linker gives you an unresolved external reference, you may have failed to declare an external _fastcall function properly.

This Page is Blank
Do Not Print

Section 7
Managing Memory

When developing advanced applications, it is important to consider how your program manipulates data and code in memory, and then select an appropriate memory model. Carefully-considered memory-management strategies make programs run faster and use less memory because 80x86 architecture is based on 64K memory segments. To manage memory effectively, you need to understand how the different types of pointers work in and across memory segments.

This section covers these topics:

- Pointers and 64K Segments
- Using Standard Memory Models
- Mixing Memory Models
- Customizing Memory Models
- Using Based Variables

Pointers and 64K Segments

CTOS workstations use Intel 8086, 80186, 80286, or 80386 processors (collectively called the 80x86 family). These processors have a segmented architecture, which means they treat memory as a series of segments that occupy up to 64K of memory each. The 64K limit exists because the 80x86 registers are 16 bits (2 bytes) wide, so a single register can address 65,536 (64K) memory locations only.

Every program has at least two parts: code (function definitions) and data (variables and constants). The 80x86 CS register holds the base for the code segment; the DS register holds the base for the data segment. Two other segment registers are available: the stack segment register (SS) and the extra segment register (ES). (The 80386 has the additional segment registers FS and GS, but the C compiler does not generate code to use them.)

As a program runs, it refers to elements of code or data with addresses. These addresses can be stored in variables called pointers, which you can use to directly access memory locations.

Pointers can fit into either 16 or 32 bits. Objects referenced in the same segment require 16-bit pointers only. Objects referenced in other segments require 32-bit pointers: 16 bits for the segment location, and 16 bits for the offset within the segment.

CTOS Microsoft C supports four types of pointers:

- Near
- Far
- Huge
- Based

Near Pointers

Near pointers contain 16 bits, and point to objects within a single 64K segment. These objects are called near objects, and accessing near objects is called near addressing.

If you do not explicitly specify a memory model, the compiler defaults to the small model, which allots up to 64K for code and another 64K for data, as shown in Figure 7-1.

When a small-model program runs, segment registers CS and DS never change. Thus, all code pointers and all data pointers contain 16 bits because they remain within the 64K range.

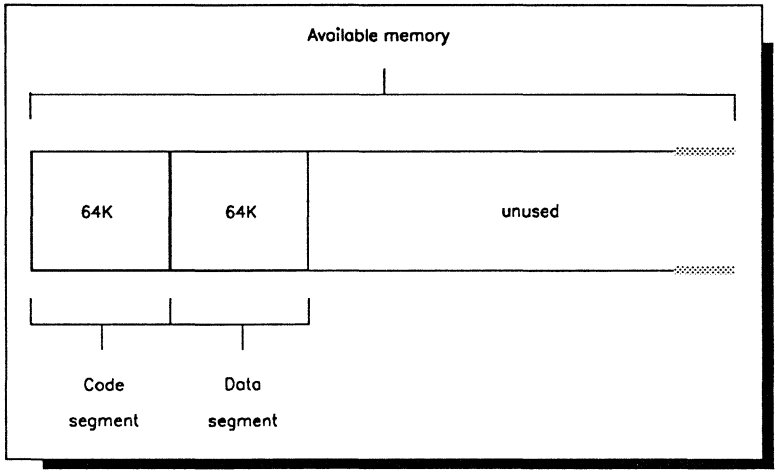


Figure 7-1. Anatomy of Small-Model Program

Far Pointers

Far pointers contain 32 bits, and point to objects and segments anywhere in memory, extending beyond a single 64K segment. These objects are called far objects, and accessing far objects is called far addressing.

Far addressing has the advantage that your program can address any available memory location.

Two disadvantages of far pointers is that they use more memory (four bytes instead of two), and using them takes more time (when assigning, modifying, or otherwise accessing values).

Allowing either code or data to expand beyond 64K makes programs larger and slower.

Huge Pointers

Huge pointers are similar to far pointers in that they contain 32 bits. Huge pointers differ, however, in that they apply to data pointers only, not to code pointers, and allow data objects (arrays) to extend beyond a single 64K segment.

Huge pointers permit this size extension by performing pointer arithmetic on all 32 bits of the data item's address. This allows items referenced by huge pointers to span more than one segment, which can be advantageous for large arrays.

However, while huge pointers allow arrays larger than 64K, huge pointer arithmetic executes slower than far pointer arithmetic. Far pointers assume that code and data objects lie completely within the segment in which they start. As a result, pointer arithmetic operates on the offset address only, making far pointer arithmetic faster.

The following fragment illustrates the difference between far and huge pointers, by incrementing both hp and fp:

```
int _huge *hp;
int _far *fp;
.
.
.
hp++;
fp++;
```

In this example, only the offset part of the far pointer (a 16-bit value) is incremented. The offset of the huge pointer is incremented and, if the offset overflows, the segment (or selector) portion is incremented appropriately. In protected-mode programs, the huge selector is set to the value of the next selector. In real mode, the segment portion of the 32-bit address is incremented to point to the next 64K block of memory.

Huge pointer arithmetic executes more slowly than far pointer arithmetic: you gain the use of larger arrays by paying a price in execution speed.

Based Addressing

A based pointer operates as a 16-bit offset from a selector or segment base that you specify. In this respect, based addressing differs from near, far, or huge addressing, because you name the base instead of the compiler. When you declare near, far, and huge variables, the compiler and linker automatically manage details such as allocating memory and keeping track of segments.

Standard Memory Models

To eliminate having to declare pointers as near or far, you can set all pointers to one size by selecting a standard memory model. In fact, selecting a model applies your choice of default pointer size to all variables in a program.

Using memory models has advantages and disadvantages. One advantage is that you have to specify compiler storage allocation for code and data once only. Another advantage is that you do not have to use Microsoft-specific keywords such as _near and _far. Thus, memory models are best for writing portable code.

One disadvantage, however, is that because standard memory models make global assumptions about the program's execution environment, they do not always produce the most efficient code.

The five memory models are shown in Table 7-1.

Table 7-1. Memory Models

Model	Maximum Code	Total Data	Memory Data Arrays
Small	64K	64K	64K
Medium	No limit	64K	64K
Compact	64K	No limit	64K
Large	No limit	No limit	64K
Huge	No limit	No limit	No limit

When you choose a model, the compiler inserts the name of the corresponding C run-time combined library in the object file. Each memory model has its own library except huge (which uses the large-model library).

Text that follows covers these topics:

- Selecting Memory Models
- Limitations on Code Size and Data Size
- How to Use the Huge Memory Model
- Null Pointer Sizing Issues

Selecting Memory Models

You can select memory models either from Programmer's WorkBench (PWB), or on the command line. If you do not specify a memory model, the compiler defaults to the small model, which is adequate for many small to mid-sized programs.

Selecting from PWB

If you are compiling from PWB, open the Options menu, and select Build Options. The available memory models appear in the upper left corner. Select one of the standard models, or select Customized and type in the options for a customized model.

Selecting from the Command Line

You can choose a memory model by including an option on the command line. For example, to compile CLICK.c as a compact-model program, type:

```
CL
  [Args]     /AC CLICK.c
```

The /AC option selects the compact memory model. The five options and four libraries are listed below:

Table 7-2. Memory Model Options and Libraries

Option	Memory Model: Library
/AS	Small Model: SLIBCP.LIB
/AM	Medium Model: MLIBCP.LIB
/AC	Compact Model: CLIBCP.LIB
/AL	Large Model: LLIBCP.LIB
/AH	Huge Model: LLIBCP.LIB

Notes: *When using MCBIND, you must specify the library that matches your compiler option in the command form. In addition, you must always specify LIBCEXT.LIB.*

Unlike Microsoft LINK, MLINK and MCBIND do not search the library specified in the object file, but MLINK does default to the correct C libraries if you do not specify /c when compiling.

Limitations on Code Size and Data Size

When writing a program in CTOS Microsoft C, keep in mind two limitations that apply to all five memory models:

- No single source module can generate 64K or more of code. You must break large programs into modules and link their individual .Obj files to create the Run file.
- No single data item can exceed 64K unless it appears in a huge-model program or it has been declared with the _huge keyword.

How to Use The Huge Memory Model

The huge memory model is nearly identical to the large model. The only difference is that the huge model permits individual arrays to exceed 64K in size. For example, an int uses two bytes, so an array of 40,000 integers, occupying 80,000 bytes of memory, would be permitted in the huge model. All other models limit each array, structure, or other data object to no more than 64K.

Note: *Automatic arrays cannot be declared huge. Only static arrays and arrays occupying memory allocated by the halloc function can be huge.*

Although the huge model lifts the limits on arrays, some size restrictions do apply. To maintain efficient addressing, no individual array element is allowed to cross a segment boundary. This affects size in the following ways:

- No single element of an array can be larger than 64K. An array can be larger than 64K, but its individual elements cannot.
- For any array larger than 128K, all elements must have a size in bytes equal to a power of 2: 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on. If the array is 128K or smaller, its elements can be any size, up to and including 64K.

Pointer arithmetic changes within the huge model, as well. In particular, the sizeof operator may return an incorrect value. The ANSI standard for C defines the value returned by sizeof to be of type size_t (which, in CTOS Microsoft C, is an unsigned int). The size in bytes of a huge array is an unsigned long value, however. To find the correct value, you must use a type cast:

```
(unsigned long)sizeof(monster_array)
```

Similarly, the C language defines the result of subtracting two pointers as ptrdiff_t (a signed int in CTOS Microsoft C). Subtracting two huge pointers yields a long value. CTOS Microsoft C gives the correct result with the following type cast:

```
(long)(ptr1_huge - ptr2_huge)
```

When you select huge model, all extern arrays are treated as _huge. Operations on data declared as _huge can be less efficient than the same operations on data declared as _far.

Null Pointer Sizing Issues

Within the medium and compact models, code and data pointers differ in size: one is 16 bits wide and the other is 32 bits wide. When using these memory models, you should be careful when using the NULL manifest constant.

NULL represents a null data pointer. The C include files define it as:

```
#define NULL ((void *) 0)
```

In memory models where data pointers have the same size as code pointers, the actual size of a null pointer does not matter. In memory models where code and data pointers are different sizes, problems can occur. Consider this example:

```
void main()
{
  func1(NULL);
  func2(NULL);
}

  func1(char *dp)
{
  .
  .
  .
}

  func2(char (*fp)(void))
{
  .
  .
  .
}
```

In the absence of function prototypes for func1 and func2, the compiler always assumes that NULL refers to data and not code.

The example above works correctly in small, large, and huge models because, in those models, a data pointer is the same size as a code pointer. Under medium or compact model, however, main passes NULL to func2 as a null data pointer rather than as a null code pointer (a pointer to a function), which means the pointer is the wrong size.

To ensure that your code works properly in all models, declare each function with a prototype. For example, before main, include these two lines:

```
int func1(char *dp);
int func2(char (*fp)(void));
```

If you add these prototypes to the example, the code works properly in all memory models. Prototypes force the compiler to coerce code pointers to the correct size. Prototypes also enable strong type-checking of parameters.

Mixing Memory Models

In standard memory models, all data pointers and code pointers are the same size and type. A mixed memory model, however, selectively combines different types of pointers within the same program. As a result, a mixed model extends the limits of a memory model, but retains its benefits.

For example, suppose you added an array to a small-model program that pushed the data segment past the 64K limit. You could solve this problem two ways.

First, you could change the memory model from small to compact, which increases all data pointers from two to four bytes. However, the Run file size grows and execution time slows.

A second and perhaps better solution is to stay with the standard small memory model, which uses near pointers, but declare the new array as far. As a result, you end up mixing near and far pointers, creating a mixed model.

CTOS Microsoft C lets you override the standard addressing convention for a given memory model by specifying that certain items are _near, _far, _huge, or _based. These keywords are not a standard part of the C language. They are Microsoft extensions meaningful only on systems that use 80x86 microprocessors. Using these keywords affects the portability of your code.

For example, you can compile a program in the small model, but declare a certain array to be _far. At run time, the address of that array occupies four bytes. Although the program may slow slightly when accessing items in the far array, all other addressing is near. Note that all pointers to elements of an array declared as _far must also be declared as _far.

Table 7-3 lists the effects of these keywords on data pointers, code pointers, and pointer arithmetic.

Table 7-3. Addressing Declared with Microsoft Keywords

Keyword	Data	Code	Arithmetic
_near	Data reside in default data segment; 16-bit addresses	Functions reside in current code segment; 16-bit addresses	16 bits
_far	Data can be anywhere in memory, not necessarily in the default data segment; 32-bit addresses	Functions can be called from anywhere in memory; 32-bit addresses	16 bits
_huge	Data can be anywhere in memory, not necessarily in the default data segment. Individual data items (arrays) can exceed 64K in size; 32-bit addresses	Not applicable; code cannot be declared _huge	32 bits (data only)
_based	Data can be anywhere in memory, not necessarily in the default data segment; 16-bit addresses plus a known base provide the range of 32-bit addresses	Not applicable; code cannot be declared _based	16 bits (data only)-

Note: *Previous versions of the compiler accepted the keywords near, far, and huge without an initial underscore. Since the ANSI standard for C permits compiler implementors to reserve identifiers that begin with underscores, an underscore was added to these keywords to mark them as implementation-specific. To maintain compatibility with existing source code, the 6.1 version of the compiler still recognizes the obsolescent versions of these keywords.*

To mix memory models effectively, you need to avoid a few problems and follow a few rules. These problems and rules are covered under the following topics:

- Pointer Problems
- Pointer Conversions
- Rules for Declaring Near, Far, Huge, and Based Variables
- Rules for Declaring Near and Far Functions

Pointer Problems

When you declare items to be _near, _far, _huge, or _based, you can link with a standard run-time library. Be aware, however, that in some cases, the modified pointers will be incompatible with standard library functions. So watch for these problems that affect pointers:

- A library function that expects a 16-bit pointer as an argument will not function properly with modified variables that occupy 32 bits. In other words, you can cast a near pointer to a far pointer, because it adds the segment value and maintains the integrity of the address.

  If you cast a far pointer to near, however, the compiler generates a warning message because the offset may not lie within the default data segment, in which case the original far address is irretrievably lost.

- A library function that returns a pointer will return a pointer of the default size for the memory model. This is a problem only if you are assigning the return value to a pointer of a smaller size.

  For example, you may have difficulties if you compile with a model that selects far data pointers, but explicitly declare the variables to receive the return value _near.

  This warning, however, does not apply to all functions. The following functions make it easier to write mixed-model programs by providing model-independent (large model) forms for most of the standard string and memory functions. These functions can be called from any point in any program, no matter which memory model has been selected. These functions take only far pointers as arguments. Thus, any data item, near or far, in any combination, can be handled.

  The names of these functions are the same as the model-dependent forms, except they include an _f prefix. For example, _fstrlen is the model-independent version of the strlen function.

  The functions listed below are defined in the standard include file STRING.h.

_fmemccpy	_fmemcmp	_fmemcpy	_fmemset
_fmemchr	_fmemicmp	_fmemmove	

  STRING.h contains the following string functions:

_fstrcat	_fstrlen	_fstrnset	_fstrstr
_fstrchr	_fstrlwr	_fstrpbrk	_fstrtok
_fstrcmp	_fstrncat	_fstrrchr	_fstrupr
_fstricmp	_fstrncmp	_fstrrev	
_fstrcpy	_fstrncmp	_fstrset	
_fstrcspn	_fstrncpy	_fstrspn	

STRING.h contains the following string duplication functions:

_fstrdup _nstrdup

- Based pointers pose a special problem. Based pointers are passed to other functions as is (without normalization). Certain functions expect to receive based pointers, but most do not. Therefore, in most cases, you must either explicitly cast a based pointer to a far pointer or make sure that all functions that receive based pointers are prototyped.

Some run-time library functions support near, far, huge, and based variables. For example, halloc allocates memory for a huge data array.

You can always pass the value (but not the address) of a far item to a small-model library routine. For example,

```
/* Compile in small model */

 #include <stdio.h>
long _far time_val;

void main()
{
  time(&time _val);          /* Illegal far address */
  printf("%ld\n", time _val);   /* Legal value */
}
```

When using a mixed memory model, you should include function prototypes with argument-type lists to ensure that all pointer arguments are passed to functions correctly.

Pointer Conversions

Passing near or far pointers as arguments to functions can cause automatic conversions in the size of the pointer argument. Passing a pointer to an unprototyped function forces the pointer size to the larger of the following two sizes:

- The default pointer size for that type, as defined by the memory model selected during compilation

  For example, in medium-model programs, data pointer arguments are near by default, and code pointer arguments are far by default.

- The size of the type of the argument

Note that if you supply a based pointer as an argument to a function and do not specifically cast it to a far pointer type, a 16-bit offset from the base segment is passed.

If you provide a function prototype with complete argument types, the compiler performs type-checking and enforces the conversion of actual arguments to the declared type of the corresponding formal argument. However, if no declaration is present or the argument-type list is empty, the compiler converts nonbased pointer arguments automatically to the default type or the type of the argument, whichever is larger. To avoid mismatched arguments, always use a prototype with the argument types.

For example, the following program produces unexpected results in compact-model, large-model, or huge-model programs.

```
void main()
{
  int _near *x;
  char _far *y;
  int z = 1;

  test_fun(x, y, z);    /* x is coerced to far pointer in
                         * compact, large, or huge model */
}

int test_fun(int _near *ptr1, char _far *ptr2, int a)
{
  printf("Value of a = %d\n", a);
}
```

If the preceding example is compiled as a small or medium program, the size of x is 16 bits, the size of y is 32 bits, and the value printed for a is 1.

However, if the example is compiled in compact, large, or huge model, both x and y are automatically converted to far pointers when they are passed to test _fun. Since ptr1, the first parameter of test _fun, is defined as near, it receives only 16 of the 32 bits passed to it. The next parameter, ptr2, receives the remaining 16 bits passed to ptr1, plus 16 bits of the intended 32 bits passed to it. Finally, the third parameter, a, receives the final 16 bits intended for ptr2, instead of the value z in the main function.

This shifting process does not generate an error message, because both the function call and the function definition are legal. In this case the program does not work as intended, however, since the value assigned to a is not the value intended.

To pass ptr1 as a near pointer, you should include a function prototype that specifically declares this argument for test _fun as a near pointer, as shown below:

```
/* First, prototype test _fun so the compiler knows
 * in advance about the near pointer argument: */

int test _near*, char _far *, int);
main ()
{
  int _near *x;
  char _far *y;
  int z = 1;
  test _fun (x, y, z);    / * Now, x is not coerced to a far
                            * pointer; it is passed as a near
                            * pointer, no matter which memory
                            * model is used */
}

int test_fun (int _near *ptr1, char _far *ptr2, int a)
{
  printf ("Value of a = %d\n", a);
}
```

Rules for Declaring Near, Far, Huge, and Based Variables

The _near, _far, _huge, and _based keywords modify either objects or pointers to objects. When using them to declare variables, keep these rules in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarations, think of the _far keyword and the item to its right as being a single unit. For example, in the case of the declaration:

  ```
  char _far * _near *p;
  ```

  p is a near pointer to a far pointer to char, which resides in the default data segment for the memory model being used.

  By contrast, the declaration:

  ```
  char _far * _near p;
  ```

  is a far pointer to char that will always be stored in DGroup, regardless of the memory model being used.

- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item will be allocated in the default data segment (_near) or a separate data segment (_far, _huge, or _based). For example,

  ```
  char _far b;
  ```

  allocates b as an item of type char with a _far address.

- If the item immediately to the right of the keyword is a pointer, the keyword determines whether the pointer will hold a near address (16 bits), a based address (16 bits), a far address (32 bits), or a huge address (also 32 bits). For example,

  ```
  char _huge *p;
  ```

  allocates p as a huge pointer (32 bits) to an item of type char. Any arithmetic performed on the huge pointer p will affect all 32 bits. That is, the expression p++ increments the pointer as a 32-bit entity.

Rules for Declaring Near and Far Functions

You cannot declare functions as _huge or _based. The rules for using the _near and _far keywords for functions are similar to those for using them with data:

- The keyword always modifies the function or pointer immediately to its right.
- If the item immediately to the right of the keyword is a function, the keyword determines whether the function will be allocated as near or far. For example,

  ```
  char _far fun();
  ```

  defines fun as a function with a 32-bit address that returns a char. The function may be located in near memory or far memory, but it is called with the full 32-bit address. The _far keyword applies to the function, not to the return type.

- If the item immediately to the right of the keyword is a pointer to a function, the keyword determines whether the function will be called using a near (16-bit) or far (32-bit) address. For example,

  ```
  char ( _far *pfun)();
  ```

  defines pfun as a far pointer (32 bits) to a function returning type char.

- Function declarations must match function definitions.
- The _huge and _based keywords do not apply to functions. That is, a function cannot be huge (larger than 64K) or based. A function can return a huge data pointer to the calling function. A function can return a based pointer unless it is a pointer based on _self.

The example below declares fun1 as a far function returning type char:

```
char _far fun1(void);    /* small model */
char _far fun(void)
{
   .
   .
   .
}
```

Here, the fun2 function is a near function that returns a far pointer to type char:

```
char _far * _near fun2();    /* large model */
char _far * _near fun()
{
   .
   .
   .
}
```

The following example declares pfun as a far pointer to a function that has an int return type, assigns the address of printf to pfun, and prints "Hello world." twice.

```
/* Compile in medium, large, or huge model */

#include <stdio.h>
int ( _far *pfun)(char *, ...);

void main()
{
  pfun = printf;
  pfun("Hello world.\n");
  (*pfun)("Hello world.\n");

}
```

Customizing Memory Models

A third way to manage memory is to create customized memory models by combining different features from standard memory models. However, before creating your own nonstandard memory models, you should have a thorough understanding of C memory models and the architecture of 80x86 processors.

The /A*string* option lets you change the attributes of the standard memory models to create your own memory models. *string* contains three letters that you specify, which correspond to:

- Code pointer size
  - s short
  - l long
- Data pointer size
  - n near
  - f far
  - h huge
- Stack and data segment setup
  - d SS == DS
  - u SS != DS; DS reloaded
  - w SS != DS; DS not reloaded

Because each letter for the three specifications is unique, you can specify letters in any order after /A, but all three letters must be present.

The standard memory-model options (/AS, /AM, /AC, /AL, and /AH) can also be specified in the /A*string* form. As an example of how to construct memory models, the standard memory-model options are listed below with their /A*string* equivalents:

Standard	Custom Equivalent
/AS	/Asnd
/AM	/Alnd
/AC	/Asfd
/AL	/Alfd
/AH	/Alhd

For example, you might want to create a huge-compact model. This model would allow huge data items but only one code segment. The option for specifying this model would be /Ashd.

This subsection covers these topics:

- Sizing Code Pointers
- Sizing Data Pointers
- Setting Up Segments
- Library Support for Customized Memory Models
- Setting the Data Threshold
- Naming Modules and Segments
- Specifying Code and Data Segments

Sizing Code Pointers

Within a customized memory model, you must choose whether code pointers are short or long:

Option	Size
/Asxx	Short (near) code pointers
/Alxx	Long (far) code pointers

The /As (short) option tells the compiler to generate near 16-bit pointers and addresses for all functions. This is the default for small- and compact-model programs.

The /Al (long) option tells the compiler to use far 32-bit pointers and addresses to address all functions. Far pointers are the default for medium-, large-, and huge-model programs.

Sizing Data Pointers

Data pointers can be near, far, or huge:

Option	Size
/Axnx	Near data pointers
/Axfx	Far data pointers
/Axhx	Huge data pointers

The /An (near) option tells the compiler to use 16-bit pointers and addresses for all data. This is the default for small- and medium-model programs.

The /Af (far) option specifies that all data pointers and addresses are 32 bits. This is the default for compact- and large-model programs.

The /Ah (huge) option specifies that all data pointers and addresses are far (32-bit) and that arrays are permitted to extend beyond a 64K segment. This is the default for huge-model programs.

With far data pointers, no single data item can be larger than a segment (64K) because address arithmetic is performed only on 16 bits (the offset portion) of the address. When huge data pointers are used, individual data items can be larger than a segment (64K) because address arithmetic is performed on both the segment and the offset.

Setting Up Segments

Within a customized model, you can choose to make the stack segment (SS) equal the data segment (DS), in which case they overlap:

Option	Effect
/Axxd	SS == DS
/A[xx]u	SS != DS; DS reloaded on function entry
/A[xx]w	SS != DS; DS not reloaded on function entry

Option /Ad

Option /Ad tells the compiler that segment addresses stored in SS and DS registers are equal. Thus, the stack segment and the default data segment are combined into a single segment. This is the default for all standard-model programs. In small- and medium-model programs, the stack plus all data must occupy less than 64K. As a result, all data items are accessed with 16-bit offsets from the segment address in the SS and DS registers.

In compact-, large-, and huge-model programs, initialized global and static data are placed in the default data segment up to a certain threshold. The compiler stores the address of this segment in the DS and SS registers. All pointers to data, including pointers to local data (the stack), are full 32-bit addresses.

This is important to remember when passing pointers as arguments in multiple-segment programs. Although you may have more than 64K of total data in these models, no more than 64K of data can occupy the default segment. The /Gt and /ND options control allocation of items in the default data segment if a program exceeds this limit.

Option /Au

Option /Au tells the compiler that the stack segment does not necessarily coincide with the data segment. In addition, it adds the _loadds attribute to all functions within a module, causing the compiler to generate code to load the DS register with the correct value prior to entering the function body.

To name data segments other than the default, combine options /Au and /ND. When /Au is combined with /ND, the address in the DS register is saved upon entry to each function, and the new segment value for the module in which the function was defined is loaded into the DS register. The previous DS value is restored on exit from the function. Therefore, only one data segment is accessible at any given time. Using the same data segment name with the /ND option for more than one source module combines the individual data segments into a single segment.

Careful program design is critical to successfully using the /ND and /Au options. Since the DS register is loaded with the data segment specified, functions that expect DS to have the same value as SS will not work properly when called from modules compiled with the /ND option. All C library functions that reference static and global data expect DS to have the same value as SS (the value of the DGroup data segment). The /ND option is most commonly used for data-only modules.

To cause DS to be set equal to SS prior to all calls to _cdecl functions, combine /Gu with /ND.

If a standard memory-model option precedes it on the command line, the /Au option can be specified without any letters indicating data pointer or code pointer sizes. The program uses a standard memory model, but different segments are set up for the stack and data segments.

Option /Aw

Option /Aw, like /Au, causes the compiler to assume that the stack segment is separate from the data segment. The compiler does not automatically load the DS register at each function entry point. The /Aw option is useful for creating applications that interface with an operating system, or with programs running at the operating-system level. The operating system or the program running under the operating system actually receives the data intended for the application program and places that data in a segment. The operating system or program must then load the DS register with the segment address for the application program.

As with the /Au option, the /Aw option can be specified without data pointer and code pointer letters if a standard memory-model option precedes it on the command line. In such a case, the program uses the specified memory model just as with /Au, but the DS register is not reloaded at each function entry point.

Even though /Au and /Aw indicate that the stack may be in a separate segment, the stack's size is still fixed at the default size unless you override this by using the CTOS Linker's Stack Size argument.

If you choose one of the options that specifies that the stack segment is not equal to the data segment (SS != DS), you cannot pass the address of frame variables as arguments to functions that take near pointers. That is, in small and medium models, you cannot pass the address of a local variable (which is allocated on the stack) as an argument, because the receiving function will assume the pointer is relative to the data segment. However, the receiving function could solve this problem by declaring the pointer to be the following:

```
based( _segname("_STACK"))
```

Another solution would be to cast the pointer to a far pointer in both locations as follows:

```
/* Call func with an explicit cast to far */
func((char far *)frame _var);
.
 .

void func(char far *formal _var)
```

Library Support for Customized Memory Models

Most C programs make function calls to the routines in the C run-time library. When you write mixed-model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is linked. Table 7-4 shows the libraries from which to extract the start-up routine for each customized memory model.

Table 7-4. Start-Up Routines for Customized Memory Models

Memory-Model Option	From Library
/Asnx; /AS plus /Ax	SLIBCP.LIB
/Asfx; /Ashx; /AC plus /Ax	CLIBCP.LIB
/Alnx; /AM plus /Ax	MLIBCP.LIB
/Alfx; /Alhx; /AL plus /Ax; /AH plus /Ax	LLIBCP.LIB

The /Ax option represents either /Au or /Aw.

Setting the Data Threshold

The /Gt option causes all initialized data items whose size is greater than number bytes to be allocated to a new data segment. The syntax is:

`/Gt[number]`

When specifying number, it must appear without spaces after the /Gt option. When number is omitted, the default threshold value is 256. When the /Gt option is omitted, the default threshold value is 32,767.

The /Gt option applies only to compact-, large-, and huge-model programs, since small- and medium-model programs have only one data segment. This option is particularly useful with programs that have more than 64K of initialized static and global data in small data items, because otherwise you run out of memory in the default data segment and cannot link the program. The /Gt option has no effect on uninitialized global data.

Naming Modules and Segments

You can use the following options to name modules and segments:

Option	Effect
/NM *modulename*	Names the module
/NT *textsegment*	Names the code segment
/ND *datasegment*	Names the data segment

Module is another name for an object file created by the C compiler from a single source file. Every module has a name. The compiler uses this name in error messages if problems are encountered during processing. The module name is usually the same as the source-file name. You can change this name using the /NM (name module) option. The new *modulename* can include any combination of letters and digits. The space between /NM and *modulename* is optional.

Every module has at least two segments: a code segment (sometimes called the text segment) containing the program instructions, and a data segment containing the program data.

The compiler normally creates the code and data segment names. The default names depend on the memory model chosen for the program. For example, in small-model programs the code segment is named _TEXT and the data segment is named _DATA.

Table 7-5 summarizes the naming conventions for code and data segments.

Table 7-5. Segment-Naming Conventions

Model	Code	Data	Module
Small	_TEXT	_DATA	---
Medium	*module*_TEXT	_DATA	*filename*
Compact	_TEXT	_DATA	*filename*
Large	*module*_TEXT	_DATA	*filename*
Huge	*module*_TEXT	_DATA	*filename*

In memory models that contain multiple data segments (compact, large, and huge), _DATA is the name of the default data segment. Other data segments have unique private names. You can override the default names with the options /NT (name text) and /ND (name data).

The /ND option is commonly used to create and compile modules that contain data only. Such modules can be accessed from other parts of the program by declaring their variables as external.

If you change the name of the default data segment with /ND, your program must load the DS register with the segment selector of your named data segment before it accesses it. You must therefore compile your program either with the /Astring form of the memory-model option and the /Au option for the segment setup, or with the /A option for a standard memory model followed by /Au. For example,

```
CL
  [Args]     /AS /Au /ND DATA1 PROG1.c
```

The /Au option forces the compiler to generate code to load DS with the correct data-segment value on entry to the code.

All modules whose data segments have the same name have these segments combined into a single segment named DATA1 at link time.

The functions in the C run-time libraries that rely on the default data segment being named _DATA will fail if you use the /ND option to rename the default data segment. You must use the /Gu option with /ND to reset the DS register to the _DATA segment before calling C-library functions.

Specifying Code and Data Segments

The following pragmas give you more control over the distribution of functions and data:

- `#pragma alloc_text(`*`textsegment, function1`*`[,`*`function2`*`]...)`
- `#pragma same_seg(`*`variable1`*`[,`*`variable2`*`]...)`

The alloc_text pragma lets you name the segment in which particular functions are allocated. The same_seg pragma provides information the compiler can use to generate better code by assuming that the specified variables are in the same segment.

If you use overlays or swapping techniques to handle large programs, alloc_text allows you to tune the contents of their code (text) segments for maximum efficiency. The alloc_text pragma must appear before the definitions of any of the specified functions and after the declarations of these functions. Functions referenced in an alloc_text pragma should be defined in the same module as the pragma. If this is not done, and an undefined function is later compiled into a different code segment, the error may not be caught.

The same_seg pragma tells the compiler to assume that the specified external variables are allocated in the same data segment. You are responsible for making sure that these variables are put in the same data segment; one way to do this is to specify the /ND option when you compile the program.

The same_seg pragma must appear before a specified variable appears in the executable code but after the variable is declared. Variables specified in a same_seg pragma must be explicitly declared with extern storage class, and they must either be explicitly declared with the _far keyword or assumed to be far because the memory model is compact, large, or huge.

Using Based Variables

Whenever you declare a near, far, or huge data variable, the compiler handles the details of where the pointer is stored and how memory is allocated.

With based variables, however, you name a base that specifies where in memory the data resides. This subsection explains how and why to include based variables in your programs, and covers these topics:

- Advantages of Based Pointers
- Based Pointer Keywords
- Declaring Based Variables

Advantages of Based Pointers

The advantage of near pointers is that they occupy only two bytes in memory. The disadvantage is that they can only address the 65,536 locations in the default data segment.

The advantage of far pointers is that they remove the addressing limit; they can point to any address. The disadvantage is that they need twice as much memory for each pointer: four bytes instead of two.

Based pointers are as small as near pointers but as flexible as far pointers, they enjoy both of the benefits and neither of the drawbacks of the other pointers. Like a near pointer, a based pointer occupies only two bytes. Like a far pointer, a based pointer can point anywhere in memory. You must, however, provide some extra information about where the base is.

Based pointers do not need the two extra bytes used by far pointers to name the segment. When necessary, the compiler generates the code to switch segments to the new base.

If you write programs that use many far pointers, you may be able to save memory by converting the four-byte far pointers to two-byte based pointers.

Near pointers always give you the fastest code. Far and huge pointers always give you the slowest.

Based pointers lie somewhere in between. When a function accesses a group of based pointers that have the same base, the extra segment register (ES) may be loaded only once. If you enable full optimization and use the same base for your based pointers, in many cases they will be faster than far pointers and nearly as fast as near pointers.

Based pointers can give you access to the code, data, stack, or constant segments. For example, if you want to use the small memory model, but the data requires more than 64K, you can store some of the constant strings in the code segment (instead of the constant segment).

Another benefit of based pointers is that you can swap data from disk to memory or from one area of memory to another. If a series of variables has the same base, you can easily move the block of memory around without having to reinitialize the variables' address.

Based Pointer Keywords

The keywords in Table 7-6 are used with _based pointer declarations and statements.

Table 7-6. Keywords

Keyword	Use
_based(*base-expression*)	Qualifies a declaration to indicate that a variable is based. In the same class as _near, _far, and _huge. It is always followed by a base expression in parentheses.
_segment	New data type that holds a memory segment address. In the same class as char, int, and float.
_segname("*segmentname*")	The name of the segment.
_self	A base expression that names itself as a base.
:>	The base operator that combines a segment and an offset to produce an effective address.

Declaring Based Variables

The _based keyword is similar in most respects to the related keywords _near and _far. You can use it anywhere that _near or _far might appear.

The _based keyword is always followed by a base in parentheses. For example,

```
char _based(seg1) *bp
```

means that bp is a based pointer to char. In this example, the base is the variable seg1.

Information that follows describes these types of based expressions:

- Variables and Pointers Based on a Segment Constant
- Pointers Based on a Segment Variable
- Pointers Based on a Pointer
- Pointers Based on Void
- Pointers Based on a Self Segment

Variables and Pointers Based on a Segment Constant

One way to declare a based variable is to give it a segment constant as a base. Four segments are predefined in CTOS Microsoft C:

Segment	Definition
_CODE	The default code segment
_CONST	The constant segment for strings such as "This is a constant string."
_DATA	The default data segment
_STACK	The stack segment

The _segname keyword marks the name of a segment. It is always followed by parentheses and a string, as in the example below:

```
/* Compile in Small Model */

#include <stdio.h>
#include <malloc.h>

char_based(_segname("_CODE")) mystring[ ]= "A code-based
string.\n";
int _based(_segname("_CODE")) ib = 12345;
void main()
{
 printf("%Fs %d", (char _far *)mystring, ib);
}
```

The variable mystring is declared as an array of characters based in the code segment. The variable ib is an integer (not a pointer) that is also based in the code segment.

Note that the small-model version of printf would treat mystring as a near pointer. The F in the format specifier %Fs forces the function to treat it as a far pointer and the cast to char _far* coerces the address to four bytes.

You can also name your own segments. The declaration of mystring might look like this:

```
char _based(_segname("MYSEGMENT")) mystring[ ] = "Another based
string.\n";
```

In the example above, the compiler creates a new segment called MYSEGMENT and places the string there.

Pointers Based on a Segment Variable

The _segment keyword is a new primitive type that can contain the base value of a segment. You can declare variables as type _segment, or you can coerce variables of other types to type _segment using standard C cast syntax. The key feature of variables of type _segment is that you can use them in the declaration of other _based variables. The following examples illustrate how _segment works:

```
/* a_segment can contain a segment value */
_segment a_segment;

/* The pointer based_on_segvar will always be dereferenced
 * relative to a segment base of a_segment.  */
char _based(a_segment) *based_on_segvar;

char near *near_ptr;
/* The pointer based_on_segvar will be dereferenced relative to
 * the segment base of near_ptr (which is the current value of
DS).  */
char _based((_segment)near_ptr) *based_on_segvar;

char far *far_ptr;
/* A pointer based_on_segvar will be dereferenced relative to
the
 * segment base of far_ptr.  */
char _based((_segment)far_ptr) *based_on_segvar;
```

Declaring variables as based on a segment variable allows you to group based data in the same segment.

In the following example, segvar is a variable of type _segment. The program requests memory from the heap and bases a variable there.

```
/* Compile in Small Model */
#include <malloc.h>
#include <stdio.h>
#include <string.h>

_segment segvar;
char _based(segvar) *b_string;

void main()
{
  if((segvar =_bheapseg(1000)) !=_NULLSEG)
  {
    if((b_string = _bmalloc(segvar, 20)) != _NULLOFF)
    {
        _fstrcpy((char _far *)b_string, (char _far *)"This is a
        test.\n");
        printf("%Fs", (char _far *)b_string);
        printf("Size = %d\n", sizeof b_string); /* Always 2 */
        _bfree(segvar, b_string);
     }
     else
       puts("bmalloc failed");
       _bfreeseg(segvar);
  }
     else
       puts("_bheapseg failed.");
}
```

first, the program asks for 1,000 bytes in a new based heap segment:

```
if((segvar =_bheapseg(1000)) !=_NULLSEG)
```

On failure, the _bheapseg function returns _NULLSEG (null segment). Otherwise, segvar holds the valid address of a segment.

Next, the _bmalloc function allocates 20 bytes of memory within the segment base and assigns the offset to b_string:

```
if((b_string =_bmalloc(segvar, 20)) !=_NULLOFF)
```

In this case, _NULLOFF means null offset and indicates the failure of _bmalloc. If all is well, the program continues with this code:

```
_fstrcpy((char _far *)b_string, (char _far *)"This is a
test.\n");
 printf("%Fs", (char _far *)b_string);
 printf("Size = %d\n", sizeof b_string);     /* Always 2 */
```

The standard strcpy function will not work because this is a small-model program that expects all pointers to be near. The _fstrcpy function allows you to copy to a far string. Then the string and its size are printed.

Finally, the offset memory and the segment memory are freed:

```
_bfree(segvar, b_string);_bfreeseg(segvar);
```

Pointers Based on a Pointer

A based pointer can use another pointer as its base. In the example below, the variable ip is a pointer to an integer. It serves as the base for both bp (a pointer to an integer) and cp (a pointer to a character).

Note: *Only pointer variables can be based on a pointer. Nonpointer variables (objects) cannot be based on a pointer.*

To find the actual address to which bp points, you take the address in its base (ip) and add the value in bp. For example, if ip points to location 0x2345 and bp holds a 3, then it points to 0x2348. Changing the value in the base immediately changes the addresses to which the based pointers point.

The following example illustrates pointers based on a pointer:

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

int *ip;                    /* int pointer*/
int _based(ip) *bp;        /* based on ip */
char _based(ip) *cp;

void main()
{
  int *mem1, *mem2;

  bp = (void *)0;     /* bp equals *(ip+0) */
  cp = (void *)2;     /* cp equals *(ip+2) */

  if((mem1 = (int *)malloc(100)) != NULL)
    if((mem2 = (int *)malloc(100)) != NULL)
    {
      ip = mem1;      /* ip points to mem1 */
      *bp = 5;
      strcpy((char *)cp, "String stored in mem1.");

        ip = mem2; /* ip now points to mem2 */
      *bp = 12345;
      strcpy((char *)cp, "String stored in mem2.");

      ip = mem1; /* point to mem1 */
      /* which still holds previous values */
      printf("%s *bp= %i\n", (char *)cp, *bp);

      ip = mem2; /* point to mem2 */
      /* display the values there */
      printf("%s *bp= %i\n", (char *)cp, *bp);

      free(mem2);
      free(mem1);

    }
    else puts("Second malloc failed.");
  else puts("First malloc failed.");
}
```

Two calls to malloc provide two sections of memory, whose addresses are stored in the variables mem1 and mem2. When ip is assigned one of these addresses (mem1), the pointers based on ip point somewhere within that piece of memory. When ip is assigned the address in mem2, the effective addresses of bp and cp also change.

Pointers Based on Void

A third way to declare a based pointer is to give it no base at all--to base it on void. This creates a generic pointer that acts as an offset into a segment. Such a based pointer can then be combined with a segment name to specify any address.

Note: Only pointer variables can be based on void. Nonpointer variables (objects) cannot be based on void.

To combine a segment and an offset, use the base operator. It consists of a colon and a greater-than symbol (:>). Place it between a segment value and an offset as follows:

```
segment:>offset
```

The address can be dereferenced with the indirection operator (*).

Pointers Based on a Self Segment

Another way to declare a based pointer is to use the _self keyword cast to a segment value, as in the following example:

```
typedef struct tree TREE;
struct tree
{
  int name;
  TREE _based((_segment)_self) *left;
  TREE _based((_segment)_self) *right;
};

void main()
{
  TREE _based(_segname("MYSEGMENT")) t1;
```

Any based declarations that are based on _self must apply to pointers only. Ordinary data objects cannot be self-based.

The example above declares a structure called tree and then declares t1 to be such a structure. The pointers within the structure are self-based, meaning they will point within the segment in which the tree structure is located. This is useful when the entire tree would fit into a single based segment. Note that functions cannot return pointers based on _self.

Section 8
Using the In-Line Assembler

Assembly language lets you improve program speed, reduce memory needs, and control the hardware. The in-line assembler lets you embed assembly-language instructions directly into your C source programs without extra assembly and link steps. The in-line assembler is built into the compiler--you do not need a separate assembler such as the CTOS Microsoft Macro Assembler (MASM).

This section explains how to use the CTOS Microsoft C in-line assembler, and covers these topics:

- Advantages of In-Line Assembly
- The _asm Keyword
- Using Assembly Language in _asm Blocks
- Using and Preserving Registers
- Jumping to Labels
- Calling C Functions
- Defining _asm Blocks as C Macros
- Optimizing

Advantages of In-Line Assembly

Because the in-line assembler does not require separate assembly and link steps, it is more convenient than a separate assembler. In-line assembly code can use any C variable or function name that is in scope, so it is easy to integrate with your program's C code. And because the assembly code can be mixed in-line with C statements, it can do tasks that are cumbersome or impossible in C alone.

The uses of in-line assembly include:

- Writing functions in assembly language
- Spot-optimizing speed-critical sections of code.
- Creating handler routines that require knowledge of processor states

In-line assembly is a special-purpose tool. If you plan to transport an application, you will probably want to place machine-specific code in a separate module. And because the in-line assembler does not support all of MASM's macro and data directives, you may find it more convenient to use MASM for such modules.

The _asm Keyword

The _asm keyword invokes the in-line assembler and can appear wherever a C statement is legal. It cannot appear by itself. It must be followed by an assembly instruction, a group of instructions enclosed in braces, or, at the very least, an empty pair of braces. The term _asm block here refers to any instruction or group of instructions, whether or not in braces.

Below is a simple _asm block enclosed in braces. On DOS systems, this code prints the beep character, ASCII 7.

```
_asm

{

  mov ah, 2

  mov dl, 7

  int 21h

}
```

Alternatively, you can put _asm in front of each assembly instruction:

```
_asm mov ah, 2

_asm mov dl, 7

_asm int 21h
```

Since the _asm keyword is a statement separator, you can also put assembly instructions on the same line:

```
_asm mov ah, 2 _asm mov dl, 7 _asm int 21h
```

All three examples generate the same code, but the first style--enclosing the _asm block in braces--has some advantages. The braces clearly separate assembly code from C code and avoid needless repetition of the _asm keyword. Braces can also prevent ambiguities. If you want to put a C statement on the same line as an _asm block, you must enclose the block in braces. Without the braces, the compiler cannot tell where assembly code stops and C statements begin. Finally, since the text in braces has the same format as ordinary MASM text, you can easily cut and paste text from existing MASM source files.

The braces enclosing an _asm block do not affect variable scope, as do braces in C. You can also nest _asm blocks, but the nesting does not affect variable scope.

Using Assembly Language in _asm Blocks

The in-line assembler has much in common with other assemblers. For example, it accepts any expression that is legal in MASM, and it supports all 80286 and 80287 instructions. This subsection describes the use of assembly-language features in _asm blocks, and is divided into these topics:

- Instruction Set
- Expressions
- Data Directives and Operators
- EVEN and ALIGN Directives
- Macros
- Segment References
- Type and Variable Sizes
- Comments
- The _emit Pseudoinstruction

Instruction Set

The in-line assembler supports the full instruction set of the Intel 80286 and 80287 processors. It does not recognize 80386- and 80387-specific instructions. To use 80286 or 80287 instructions, compile with the /G2 option.

Expressions

In-line assembly code can use any MASM expression: any combination of operands and operators that evaluates to a single value or address.

Data Directives and Operators

Although an _asm block can reference C data types and objects, it cannot define data objects with MASM directives or operators. Specifically, you cannot use the definition directives DB, DW, DD, DQ, DT, and DF, or the operators DUP or THIS. Nor are MASM structures and records available. The in-line assembler does not accept the directives STRUC, RECORD, WIDTH, or MASK.

EVEN and ALIGN Directives

While the in-line assembler does not support most MASM directives, it does support EVEN and ALIGN. These directives put NOP (no operation) instructions in the assembly code as needed to align labels to specific boundaries. This makes instruction-fetch operations more efficient for some processors.

Macros

The in-line assembler is not a macro assembler. You cannot use MASM macro directives (MACRO, REPT, IRC, IRP, and ENDM) or macro operators (<>, !, &, %, and .TYPE). An _asm block can use C preprocessor directives, however. For details, refer to the heading Using C in _asm Blocks.

Segment References

You must refer to segments by register rather than by name (the segment name _TEXT is invalid, for instance). Segment overrides must use the register explicitly, as in ES:[BX].

Type and Variable Sizes

The LENGTH, SIZE, and TYPE operators have a limited meaning in in-line assembly. You cannot use them with the DUP operator (because you cannot define data with MASM directives or operators). You can, however, use them to find the size of C variables or types:

- The LENGTH operator can return the number of elements in an array. It returns the value 1 for nonarray variables.
- The SIZE operator can return the size of a C variable. A variable's size is the product of its LENGTH and TYPE.
- The TYPE operator can return the size of a C type or variable. If the variable is an array, TYPE returns the size of a single element of the array.

For instance, if your program has the eight-element int array

```
int arr[8];
```

the following C and assembly expressions yield the size of arr and its elements:

_asm	C	Size
LENGTH arr	sizeof(ar)/sizeof(arr[0])	8
SIZE arr	sizeof (arr)	16
TYPE arr	size14(arr[0])	2

Comments

Instructions in an _asm block can use assembly-language comments:

```
_asm mov ax, offset buff ; Load address of buff
```

Because C macros expand into a single logical line, avoid using assembly-language comments in macros. An _asm block can also contain C-style comments.

The _emit Pseudoinstruction

The _emit pseudoinstruction is similar to the DB directive of MASM. It allows you to define a single immediate byte at the current location in the current text segment. However, _emit can define only one byte at a time, and it can only define bytes in the text segment. It uses the same syntax as the INT instruction.

One use for _emit is to define 80386-specific instructions, which the in-line assembler does not support. The following fragment, for instance, defines the 80386 CWDE instruction:

```
/* Assumes 16-bit mode */
#define cwde _asm _emit 0x66 _asm _emit 0x98
   .
   .
   .
_asm
{
        cwde
}
```

Debugging and Listings

Programs containing in-line assembly code can be symbolically debugged with the CodeView debugger. You must, however, compile with option /Zi. If you compile with switch /c, you must also specify CodeView in the [Debugger] field for MCBIND. If you do not compile with /c, linking occurs automatically with MLINK, which passes the correct value to the CTOS Linker.

Within CodeView, you can set breakpoints on both C and assembly-language lines. If you enable mixed assembly and C mode, you can display both the source and disassembled form of the assembly code.

Note that putting multiple assembly instructions or C statements on one line can hamper debugging with CodeView. In source mode, the CodeView debugger lets you set breakpoints on a single line, but not on individual statements on the same line. The same principle applies to an _asm block defined as a C macro, which expands to a single logical line.

If you create a mixed source and assembly listing with the /Fc compiler option, the listing contains both the source and assembly forms of each assembly-language line. Macros are not expanded in listings, but they are expanded during compilation.

Using C in _asm Blocks

Because in-line assembly instructions can be mixed with C statements, they can refer to C variables by name and use many other elements of C. An _asm block can use the following C language elements:

- Symbols, including labels and variable and function names
- Constants, including symbolic constants and enum members
- Macros and preprocessor directives
- Comments (both /* */ and //)
- Type names (wherever a MASM type would be legal)
- typedef names, generally used with operators such as PTR and TYPE or to specify structure or union members

Within an _asm block, you can specify integer constants with either C notation or assembler radix notation (0x100 and 100h are equivalent, for instance). This allows you to define (using #define) a constant in C, and use it in both C and assembly portions of the program. You can also specify constants in octal by preceding them with a 0. For example, 0777 specifies an octal constant.

This heading covers the following topics:

- Using Operators
- Using C Symbols
- Accessing C Data
- Writing Functions

Using Operators

An _asm block cannot use C-specific operators, such as the << operator. However, operators shared by C and MASM, such as the * operator, are interpreted as assembly-language operators.

For instance, outside an _asm block, square brackets ([ ]) are interpreted as enclosing array subscripts, which C automatically scales to the size of an element in the array. Inside an _asm block, they are seen as the MASM index operator, which yields an unscaled byte offset from any data object or label (not just an array). The following code illustrates the difference:

```
int array[10];
_asm mov array[6], bx ;    Store BX at array+6 (not scaled)

array[6] = 0;              /* Store 0 at array+12 (scaled) */
```

The first reference to array is not scaled, but the second is. Note that you can use the TYPE operator to achieve scaling based on a constant. For instance, the following statements are equivalent:

```
_asm mov array[6 * TYPE int], 0 ; Store 0 at array + 12

array[6] = 0;                     /* Store 0 at array + 12 */
```

Using C Symbols

An _asm block can refer to any C symbol in scope where the block appears. (C symbols are variable names, function names, and labels--in other words, names that are not symbolic constants or enum members.)

A few restrictions apply to the use of C symbols:

- Each assembly-language statement can contain only one C symbol. Multiple symbols can appear in the same assembly instruction only with LENGTH, TYPE, and SIZE expressions.
- Functions referenced in an _asm block must be declared (prototyped) earlier in the program. Otherwise, the compiler cannot distinguish between function names and labels in the _asm block.

- An _asm block cannot use any C symbols with the same spelling as MASM reserved words (regardless of case). MASM reserved words include instruction names such as PUSH and register names such as SI.
- Structure and union tags are not recognized in _asm blocks.
- Although the compiler supports C-variable referencing in _asm blocks, it does not automatically generate correct addressing for variables in something other than the default data segment. The following example illustrates this point.

```
#include <stdio.h>

unsigned short based_in_defseg = 0;

unsigned short _based(_segname("farseg")) based_in_farseg = 0;

unsigned char msg[] =
"\nbased_in_defseg=%x\nbased_in_farseg=%x\n";

void main(void)
{
    based_in_defseg = 0xface;
    based_in_farseg = 0x1234;
    printf(msg, based_in_defseg, based_in_farseg);

    _asm
    {
#if defined _NOT_USING_CORRECT_BASE_
        mov     based_in_defseg, 0x1234
        mov     based_in_farseg, 0xface
#else
        mov     based_in_defseg, 0x1234
        mov     ax, seg based_in_farseg
        mov     es, ax
        mov     es:based_in_farseg, 0xface
#endif
    }
    printf(msg, based_in_defseg, based_in_farseg);
}
```

Accessing C Data

A great convenience of in-line assembly is the ability to refer to C variables by name. An _asm block can refer to any symbols-including variable names-that are in scope where the block appears. For instance, if the C variable var is in scope, the instruction:

```
_asm mov ax, var
stores the value of var in AX.
```

If a structure or union member has a unique name, an _asm block can refer to it using only the member name, without specifying the C variable or typedef name before the period (.) operator. If the member name is not unique, however, you must place a variable or typedef name immediately before the period (.) operator. For instance, the following structure types share same_name as their member name:

```
struct first_type
{
  char *weasel;
  int same_name;
};
struct second_type
{
  int wonton;
  long same_name;
};
```

If you declare variables with the types:

```
struct first_type hal;
struct second_type oat;
```

then all references to the member same_name must use the variable name, because same_name is not unique. But the member weasel has a unique name, so you can refer to it using only its member name:

```
_asm
{
  mov bx, OFFSET hal
  mov cx, [bx]hal.same_name ; Must use 'hal'
  mov si, [bx].weasel       ; Can omit 'hal'
}
```

Note that omitting the variable name is merely a coding convenience. The same assembly instructions are generated whether or not it is present.

Writing Functions

If you write a function with in-line assembly code, it is easy to pass arguments to the function and return a value from it. The following examples compare a function first written for a separate assembler and then rewritten for the in-line assembler. The function, called power2, receives two parameters, multiplying the first parameter by 2 to the power of the second parameter. Written for a separate assembler, the function might look like this:

```
; POWER.ASM
; Compute the power of an integer
;


        PUBLIC _power2

_TEXT SEGMENT WORD PUBLIC 'CODE'
_power2 PROC
 push bp              ; Save BP
 mov  bp, sp          ; Move SP into BP so we can refer
                      ; to arguments on the stack
 mov  ax, [bp+4]      ; Get first argument
 mov  cx, [bp+6]      ; Get second argument
 shl  ax, cl          ; AX = AX * (2 ^ CL)
 pop  bp              ; Restore BP
 ret                  ; Return with sum in AX

_power2 ENDP
_TEXT   ENDS
        END
```

Since it is written for a separate assembler, the function requires a separate source file and assembly and link steps. C function arguments usually are passed on the stack, so this version of the power2 function accesses its arguments by their positions on the stack. (Note that the MODEL directive, available in MASM and some other assemblers, also allows you to access stack arguments and local stack variables by name.)

The POWER2.c program below writes the power2 function with in-line assembly code:

```
/* POWER2.c */
#include <stdio.h>

int power2(int num, int power);

void main(void)
{
  printf("3 times 2 to the power of 5 is %d\n",
  power2(3, 5));
}

int power2(int num, int power)
{
    _asm
    {
      mov ax, num   ; Get first argument
      mov cx, power ; Get second argument
      shl ax, cl    ; AX = AX * (2 to the power of CL)
    }
    /*  Return with result in AX */
}
```

The in-line version of the power2 function refers to its arguments by name and appears in the same source file as the rest of the program. This version also requires fewer assembly instructions. Since C automatically preserves BP, the _asm block does not need to do so. It can also dispense with the RET instruction, since the C part of the function performs the return.

Because the in-line version of power2 does not execute a C return statement, it causes a harmless warning if you compile at warning levels 2 or higher:

```
warning C4035: 'power2' : no return value
```

The function does return a value, but the compiler cannot tell that in the absence of a return statement. Simply ignore the warning in this context.

Using and Preserving Registers

In general, you should not assume that a register will have a given value when an _asm block begins. An _asm block inherits whatever register values happen to result from the normal flow of control.

If you use the _fastcall calling convention, the compiler passes function arguments in registers instead of the stack. This can create problems in functions with _asm blocks, since a function has no way to tell which parameter is in which register. If the function happens to receive a parameter in AX and immediately stores something else in AX, the parameter is lost. In addition, you must preserve the CX and ES registers in any function declared with _fastcall.

To avoid such register conflicts, do not use the _fastcall convention for functions that contain an _asm block. If you specify the _fastcall convention globally with the /Gr compiler option, declare every function containing an _asm block with _cdecl. (The _cdecl attribute tells the compiler to use the normal C calling convention for that function.) If you are not compiling with /Gr, avoid declaring the function with the _fastcall attribute.

You may notice in the POWER2.c example in under the heading Writing Functions, that the power2 function does not preserve the value in the AX register. When you write a function in assembly language, you do not need to preserve the AX, BX, CX, DX, ES, and flags registers. However, you should preserve any other registers you use (DI, SI, DS, SS, SP, and BP).

Note: *If your in-line assembly code changes the direction flag using the STD or CLD instructions, you must restore the flag to its original value.*

The POWER2.c example under the heading Writing Functions also shows that functions return values in registers. This is true whether the function is written in assembly language or in C.

If the return value is short (a char, int, or near pointer), it is stored in AX. The POWER2.c example returned a value by terminating with the desired value in AX.

If the return value is long, store the high word in DX and the low word in AX. To return a longer value (such as a floating-point value), store the value in memory and return a pointer to the value (in AX if near or in DX:AX if far).

Assembly instructions that appear in-line with C statements are free to alter the AX, BX, CX, and DX registers. C does not expect these registers to be maintained between statements, so you do not need to preserve them. The same is true of the SI and DI registers, with some exceptions. You should preserve the SP and BP registers unless you have some reason to change them (to switch stacks, for instance).

Jumping to Labels

Like an ordinary C label, a label in an _asm block has scope throughout the function in which it is defined (not only in the block). Both assembly instructions and C goto statements can jump to labels inside or outside the _asm block.

Unlike C labels, labels defined in _asm blocks are not case sensitive, even when used in C statements. C labels are not case sensitive in an _asm block, either. (Outside an _asm block, a C label is case sensitive as usual.) The following do-nothing code shows all the permutations:

```
void func(void)
{
  goto C_Dest;     /* legal */
  goto c_dest;     /* error */
  goto A_Dest;     /* legal */
  goto a_dest;     /* legal */

  _asm
{
     jmp C_Dest ; legal
     jmp c_dest ; legal
     jmp A_Dest ; legal
     jmp a_dest ; legal

     a_dest: ; _asm label
  }
  C_Dest:          /* C label */
  return;
}
```

Do not use C library function names as labels in _asm blocks. For instance, you might be tempted to use exit as a label, as in:

```
jne exit
  .
  .
  .
exit:
  ; More _asm code follows
```

forgetting that exit is the name of a C library function. The code does not cause a compiler error, but it might cause a jump to the exit function instead of the desired location.

As in MASM programs, the dollar symbol ($) serves as the current location counter--a label for the instruction currently being assembled. In _asm blocks, its main use is to make long conditional jumps:

```
jne $ + 5 ; next instruction is 5 bytes long
jmp farlabel
; $ + 5
  .
  .
  .
farlabel:
```

Calling C Functions

An _asm block can call C functions, including C library routines. The following example calls the printf library routine:

```
#include <stdio.h>

char format[ ] = "%s %s\n";
char hello[ ] = "Hello";
char world[ ] = "world";

void main(void)
{
   _asm
  {
     mov ax, offset world
     push a
     mov ax, offset hello
     push ax
     mov ax, offset format
     push ax
     call printf

  }
}
```

Since function arguments are passed on the stack, you simply push the needed arguments-string pointers, in the example above-before calling the function. The arguments are pushed in reverse order, so they come off the stack in the desired order. To emulate the following C statement:

```
printf(format, hello, world);
```

the example pushes pointers to world, hello, and format, in that order, then calls printf.

Defining _asm Blocks as C Macros

C macros offer a convenient way to insert assembly code into C code, but they demand extra care because a macro expands into a single logical line. To create trouble-free macros, follow these rules:

- Enclose the _asm block in braces.
- Put the _asm keyword in front of each assembly instruction.
- Use old-style C comments (/* comment */) instead of assembly-style comments (; comment) or single-line C comments (// comment).

To illustrate, the following example defines a simple macro:

```
#define BEEP _asm           \
/* Beep sound */            \
{                           \

      _asm mov ah, 2        \
      _asm mov dl, 7        \
      _asm int 21h          \
}
```

At first glance, the last three _asm keywords seem superfluous. They are needed, however, because the macro expands into a single line:

```
_asm /* Beep sound */ { _asm mov ah, 2 _asm mov dl, 7 _asm int
21h }
```

The third and fourth _asm keywords are needed as statement separators. The only statement separators recognized in _asm blocks are the newline character and _asm keyword. And since a block defined as a macro is one logical line, you must separate each instruction with _asm.

The braces are essential as well. If you omit them, the compiler can be confused by C statements on the same line to the right of the macro invocation. Without the closing brace, the compiler cannot tell where assembly code stops, and it sees C statements after the _asm block as assembly instructions.

Assembly-style comments that start with a semicolon (;) continue to the end of the line. This causes problems in macros because the compiler ignores everything after the comment, all the way to the end of the logical line. The same is true of single-line C comments (// comment). To prevent errors, use old-style C comments (/* comment */) in _asm blocks defined as macros.

An _asm block written as a C macro can take arguments. Unlike an ordinary C macro, however, an _asm macro cannot return a value. So you cannot use such macros in C expressions.

Be careful not to invoke macros of this type indiscriminately. For instance, invoking an assembly-language macro in a function declared with the _fastcall convention may cause unexpected results.

Note that some MASM-style macros can be written as C macros. For example:

```
setpage     MACRO page
            mov ah, 5
            mov al, page
            int 10h
            ENDM
```

The following code defines setpage as a C macro:

```
#define setpage(page) _asm        \
{                                 \
  _asm mov ah, 5                  \
  _asm mov al, page               \
  _asm int 10h                    \
}
```

Both macros do the same job.

Optimizing

The presence of an _asm block in a function affects optimization in a few different ways. First, as you might expect, the compiler does not try to optimize the _asm block itself. What you write in assembly language is exactly what you get.

Second, the presence of an _asm block affects register variable storage. Under normal circumstances (unless you suppress optimization with the /Od option, or use the /r option) the compiler automatically stores variables in registers. This is not done, however, in any function that contains an _asm block. To get register variable storage in such a function, you must request it with the register keyword.

Since the compiler stores register variables in the SI and DI registers, these registers represent variables in functions that request register storage. The first eligible variable is stored in SI and the second in DI. Preserve SI and DI in such functions unless you want to change the register variables.

Remember that the name of a variable declared with register translates directly into a register reference (assuming a register is available for such use). For instance, if you declare:

```
register int sample;
```

and the variable sample happens to be stored in SI, then the _asm instruction:

```
_asm mov ax, sample
```

is equivalent to:

```
_asm mov ax, si
```

If you declare a variable with register and the compiler cannot store the variable in a register, the compiler issues a warning to that effect at compile time. The solution is to remove the register declaration from that variable.

Register variables form a slight exception to the general rule that an assembly-language statement can contain no more than one C symbol. If one of the symbols is a register variable, for example:

```
register int v1;int v2;
```

then an instruction can use two C symbols, as in:

```
mov v1, v2
```

Finally, the presence of in-line assembly code inhibits the following optimizations for the entire function in which the code appears:

- Loop (/Ol)
- Global register allocation (/Oe)
- Global optimizations and common subexpressions (/Og)

These optimizations are suppressed no matter which compiler options you use.

This Page is Blank

Do Not Print

Section 9
Controlling Floating-Point Math Operations

This section describes how to control floating-point math operations, and covers the following topics:

- Declaring Floating-Point Types
- Library Support of Type long double
- Math Packages
- Floating-Point Options
- Library Considerations
- Compatibility Between Options
- Using the NO87 Environment Variable

Declaring Floating-Point Types

CTOS Microsoft C supports three floating-point types that conform to the Institute of Electrical and Electronics Engineers (IEEE) standard 754 format:

- Type float, a 32-bit floating-point quantity
- Type double, a 64-bit floating-point quantity
- Type long double, an 80-bit floating-point quantity

Using these types, you can declare:

- Variables
- Functions

Declaring Variables as Floating-Point Types

You can declare variables as float, double, or long double, depending on your application needs. The principal differences between the three types are significance, storage, and range. Table 9-1 shows the relationship between significance and storage requirements.

Table 9-1. Floating-Point Types

Type	Significant Digits	Number of Bytes
float	6-7	4
double	15-16	8
long double	19	10

Floating-point variables are represented by a mantissa that contains the value of the number, and by an exponent that contains the order of magnitude.

Table 9-2 shows the number of bits allocated to the mantissa and the exponent for each floating-point type. The most significant bit of any float, double, or long double is always the sign bit. If it is 1, the number is considered negative; otherwise, it is considered a positive number.

Table 9-2. Lengths of Exponents and Mantissas

Type	Exponent Length	Mantissa Length
float	8 bits	23 bits
double	11 bits	52 bits
long double	15 bits	64 bits

Because the exponents are stored in an unsigned form, the exponent is biased by half its possible value. For type float, the bias is 127; for type double, the bias is 1,023; for type long double, the bias is 16,383. To compute the actual exponent value, subtract the bias value from the exponent value.

The mantissa is stored as a binary fraction greater than or equal to 1 and less than 2. For types float and double, the mantissa contains an implied leading 1 in the most significant bit position. Thus, mantissas are actually 24 and 53 bits long respectively, even though the most significant bit is never stored in memory.

Instead using of the storage method just described, the floating-point package can store binary floating-point numbers as denormalized numbers. Denormalized numbers are nonzero floating-point numbers with reserved exponent values, in which the most significant bit of the mantissa is zero.

By using denormalized format, the range of a floating-point number can be extended, but at the cost of precision. You cannot control whether a floating-point number is represented in normalized or denormalized form; the floating-point package determines the representation. The floating-point packages never use denormalized form unless the exponent becomes less than the minimum that can be represented in a normalized form.

Table 9-3 shows the minimum and maximum value you can store in variables of each floating-point type. The values listed in this table apply only to normalized floating-point numbers; denormalized floating-point numbers have a smaller minimum value. Note that numbers retained in 80x87 registers are always represented in 80-bit normal form; numbers can only be represented in denormal form when stored in 32- or 64-bit floating-point variables (type float and type long).

Table 9-3. Range of Floating-Point Types

Type	Minimum Value	Maximum Value
float	1.175494351E-38	3.402823466E+38
double	2.2250738585072014E-308	1.7976931348623158E+308
long double	3.362103143112093503E-4932	1.189731495357231765E+4932

If precision is less of a concern than storage, consider using type float for floating-point variables. Conversely, if precision is the most important criterion, use type long double.

Floating-point variables can be promoted to a type of greater significance (for example, from type float to type double). Promotion often occurs when you perform arithmetic on floating-point variables. This arithmetic is always performed in as high a degree of precision as the variable with the highest degree of precision. For example, consider the following type declarations:

```
float f_short;
double f_long;
long double f_longer;

f_short = f_short * f_long;
```

In this example, the variable f_short is promoted to type double and multiplied by f_long; then the result is rounded to type float before assigned to f_short.

In the next example (using declarations from the previous example), the arithmetic is done in float (32-bit) precision on the variables; the result is then promoted to type long double.

```
f_longer = f_short * f_short;
```

Declaring Functions that Return Floating-Point Types

You can declare functions that return the floating-point types float, double, and long double. Functions that return types float or double do not place their return values in registers; they place their return values in a global location called the floating-point accumulator (_fac).

If a function that returns a floating point value is declared using the _pascal or _plm keyword to specify the PLM/Pascal calling convention, the function's return value is placed on the stack rather than in the floating-point accumulator, _fac.

Note: *Functions that return type long double always place their return values on the stack. You need not use the _pascal keyword with functions declared as long double.*

C Run-Time Library Support of Type long double

All of the CTOS Microsoft C run-time libraries support type long double. Each of the normal floating-point math functions has a special version that supports type long double. These functions have the same name as the functions that support type float and type double, except that they end with the letter "l". For example, the function that returns the absolute value of a variable of type float or type double is fabs. The long double equivalent function is fabsl. The two exceptions to this rule are the _atold and _strtodl functions.

Summary of Math Packages

The CTOS Microsoft C compiler offers a choice of the following three math packages for handling floating-point operations:

- Emulator (default)
- Math coprocessor
- Alternate math

The following descriptions of these math packages are designed to help you choose the appropriate math option for your application.

CTOS Microsoft C supports floating-point math with four basic libraries: EM.LIB for the emulator, 87.LIB for the coprocessor, *x*LIBFP.LIB to support both EM.LIB and 87.LIB , and *x*LIBFA.LIB for alternate math. (CTOS Microsoft C does not supply combined libraries like Microsoft C. For details on how to make and use combined libraries, refer to Section 5, Building Applications.)

Emulator Package

Programs created using the emulator math package automatically detect and use an 80x87 numeric coprocessor if one is installed. If no coprocessor is installed, these 80x87 instructions are carried out in software. The emulator package is the default math package. Also, the emulator math option is the option selected by default by the compiler if no other floating-point math option is specified.

Use the emulator math package to maximize accuracy on systems without math coprocessors or if your program will be run on some systems with coprocessors and some systems without coprocessors.

The emulator package performs basic operations to the same degree of accuracy as a math coprocessor. However, the emulator routines used for transcendental math functions (such as sin, cos, tan) differ slightly from the corresponding functions performed on a coprocessor. This difference can cause a slight discrepancy (usually within two bits) between the results of these operations when performed with the software emulation instead of with a math coprocessor.

When you use a math coprocessor or the emulator floating-point math package, interrupt-enable, precision, underflow, and denormalized-operand exceptions are masked by default. The remaining floating-point exceptions are unmasked. For more information about 80x87 floating-point exceptions, refer to the description of the _control87 function in online Help.

Math Coprocessor Package

The math coprocessor package uses the 80x87 math coprocessor exclusively for floating-point calculations. If you use the math coprocessor package, the target machine for your application have an 80x87 coprocessor to perform floating-point operations. This package gives you the fastest, smallest programs possible for handling floating-point math.

Alternate Math Package

The alternate math package gives you the smallest and fastest programs possible without a coprocessor. However, the program results are not as accurate as results given by the emulator package.

The alternate math package uses the same format as the IEEE standard-format numbers with less precision and weaker error checking. The alternate math package does not support infinities, NANs (Not A Number), and denormal numbers.

Compiling and Linking Floating-Point Options

You can select a method of accessing floating-point routines by specifying command-line options to the CL compiler command, or by setting options in Programmer's WorkBench (PWB). You can choose between the emulator, alternate, or math coprocessor libraries. You can also access the floating-point routines by issuing a function call (or calls), or by generating in-line 80x87 instructions to execute the floating-point operation.

The smallest and the fastest floating-point math option is the in-line math coprocessor package because the compiler generates true 80x87 coprocessor instructions. If, however, you cannot depend on the target computer having a coprocessor, you must use either the emulator or alternate math options.

To specify floating-point options on the CL command line, select an option from the list in Table 9-4. All options start with /FP. This table also lists the floating-point options you can specify when using the Programmer's WorkBench (PWB). To find these options in PWB, select **Options**, **C Compiler Options**, and then select **< Set Release Options >**.

Table 9-4. Summary of Floating-Point Compiler Options

CL Option PWB Option	Method	Coprocessor	Effect
/FPi In-Line Emulation	In-line	Uses coprocessor if present [1]	Default; larger than /FPi87, but can work without a coprocessor; most efficient way to get maximum precision without a coprocessor
/FPi87 In-Line Math Coprocessor	In-line	Requires coprocess	Smallest and fastest option available with a coprocessor
/FPc Calls to Math Coprocessor	Calls	Uses coprocessor if present [1]	Slower than /FPi, but allows use of alternate math library at link time
/FPc87 Calls to Math Coprocessor	Calls	Requires coprocessor unless library changed at link time [2]	Slower than /FPi87, but allows use of alternate math library at link time
/FPa Alternate Math	Calls	Ignores coprocessor	Fastest and smallest option available without a coprocessor, but sacrifices some accuracy for speed

[1] Set NO87 to suppress use of the coprocessor.

[2] Set NO87 to suppress use of the coprocessor if you change to the emulator library at link time.

Table 9-5 shows the libraries that support the compiler options. If you do not compile with /c, MLINK automatically links your object files with the default math libraries that support the specified compiler option. If you do compile with /c, you must separately link with either MLINK or MCBIND, and specify the libraries shown in 9-5 when you link.

Table 9-5. Floating-Point Options and Libraries

Compiler Option	Default Math Library Combinations	Optional Replacement Math Library Combinations
/FPi	EM.LIB + *x*LIBFP.LIB	87.LIB + *x*LIBFP.LIB
/FPi87	87.LIB + *x*LIBFP.LIB	
/FPc	EM.LIB + *x*LIBFP.LIB	87.LIB + *x*LIBFP.LIB
		xLIBFA.LIB
/FPc87	87.LIB + *x*LIBFP.LIB	EM.LIB + *x*LIBFP.LIB
		*x*LIBFA.LIB
/FPa	*x*LIBFA.LIB	EM.LIB + *x*LIBFP.LIB
		87.LIB + *x*LIBFP.LIB

Note: *The plus signs shown above between libraries signify that you use both libraries. The plus signs in the Optional column signify that both libraries replace the default library(s). The plus signs do not represent characters to enter on command lines.*

Optimizations such as constant propagation and constant subexpression elimination can cause some expressions to be evaluated at compile time. Such evaluations always use IEEE format and are unaffected by the floating-point option you choose. For more information about optimizing, refer to Section 6.

If you are familiar with the combined libraries used in the DOS version of Microsoft C and want to use them, you must build them. Section 5, Building Applications, describes how to build and use them.

In-Line Emulator Option (/FPi)

The in-line emulator option (/FPi) generates in-line instructions for an 80x87 coprocessor. At link time, you must specify *x*LIBFP.LIB, and either the emulator library (EM.LIB) or the math-coprocessor library (87.LIB). If you do not choose a floating-point option, the compiler uses the in-line emulator option by default.

The in-line emulator option is useful if you cannot be sure that an 80x87 coprocessor will be available on the target computer. Programs compiled using the in-line emulator option work as described below:

- If a coprocessor is present at run time, the program uses the coprocessor.
- If no coprocessor is present, the program uses the emulator.

  In this case, the in-line emulator option offers the most efficient way to get maximum precision in floating-point results.

When you use the in-line emulator option, the compiler does not generate in-line 80x87 instructions. Instead, the compiler generates software interrupts to library code, which then fixes up the interrupts to use either the emulator or the coprocessor, depending on whether a coprocessor is present. If you want true in-line 80x87 instructions, use the in-line math coprocessor option (/FPi87).

In-Line Math Coprocessor Instructions Option (/FPi87)

The in-line math coprocessor instructions option (/FPi87) instructs the compiler to place 80x87 coprocessor instructions in your code for many math operations.

If you use the in-line math coprocessor instructions option and link with the libraries 87.LIB and *x*LIBFP.LIB, an 80x87 coprocessor must be present at run time, or the program fails and the following error message displays:

```
run-time error R6002
-floating point not loaded
```

Compiling with the in-line math coprocessor instructions option results in the smallest, fastest programs possible for handling floating-point results.

Calls to Emulator Option (/FPc)

The calls to emulator option (/FPc) generate floating-point calls to the emulator library. At link time, you must specify *x*LIBFP.LIB, and either the emulator library (EM.LIB) or the math coprocessor library (87.LIB +); or specify an alternate math library (*x*LIBFA.LIB). Thus, the calls to emulator option gives you more flexibility in the libraries you can use for linking than the in-line emulator option.

Using the calls to emulator option is also recommended in the following cases:

- If you compile modules that perform floating-point operations and plan to include these modules in a library
- If you compile modules that you want to link with libraries other than the libraries provided with CTOS Microsoft C

You cannot link with an alternate math library if your program uses the intrinsic forms of floating-point library routines (that is, if you have compiled the program with the /Oi or /Ox option, selected the Generate Intrinsic Functions option from the Debug Build Options or Release Build Options dialog box in the PWB, or have specified math functions in an intrinsic pragma).

Calls to Math Coprocessor Option (/FPc87)

The calls to math coprocessor option (/FPc87) generates function calls to routines in the math coprocessor library (87.LIB and *x*LIBFP.LIB) that issue the corresponding 80x87 instructions. As with the in-line math coprocessor instructions option (/FPi87), at link time you can choose to link with the emulator library (EM.LIB and *x*LIBFP.LIB). However, /FPc offers more flexibility in choosing libraries, since you can change your mind and link with the appropriate alternate math library as well (*x*LIBFA.LIB).

The disadvantages of using the calls to math coprocessor option as opposed to the in-line coprocessor option are the following:

- Your executable size is larger because a call requires more instructions than a true coprocessor instruction.
- Your program does not execute as fast because you must issue a function call for each floating-point operation.

You cannot link with an alternate math library if your program uses the intrinsic forms of floating-point library routines (that is, if you have compiled the program with the /Oi or /Ox option or have specified math functions in an intrinsic pragma).

You must have a math coprocessor installed to run programs compiled with the /FPc option and linked with the math coprocessor library (87.LIB). Otherwise, the program fails and the following error message appears:

```
run-time error R6002
- floating point not loaded
```

Note: *Certain optimizations are not performed when you use the calls to math coprocessor option. This can reduce the efficiency of your code; also, since arithmetic of different precision can result, there may be slight differences in your results.*

Use Alternate Math Option (/FPa)

The use alternate math option (/FPa) generates floating-point calls and requires you to link with the alternate math library for the appropriate memory model (*x*LIBFA.LIB). Calls to this library provide the fastest and smallest option for code intended to run on a machine without an 80x87 coprocessor. With this option, you can also choose the emulator library (EM.LIB and *x*LIBFP.LIB) or math coprocessor library (87.LIB and *x*LIBFP.LIB) at link time.

You cannot link with an alternate math library if your program uses the intrinsic forms of floating-point library routines (that is, if you have compiled the program with the /Oi or /Ox option, selected the Generate Intrinsic Functions from the Debug Build Options or Release Build Options dialog box in PWB, or have specified math functions in an intrinsic pragma).

Library Considerations for Floating-Point Options

You may want to use libraries in addition to the default library for the floating-point option you have chosen in your compile options. For example, you may want to create your own libraries (or other collections of subprograms in object-file form), then link these libraries at a later time with object files that you have compiled using different options.

The following text describes these cases and ways to handle them. Although it assumes that you are putting your object files into libraries, the same considerations apply if you are simply using individual object files.

In-Line Instructions or Calls

When deciding on a floating-point option, you should decide whether you want to use in-line instructions. If you do, compile with the in-line math coprocessor instructions (/FPi87) or in-line emulator (/FPi) option. Otherwise, compile for floating-point function calls using the calls to math coprocessor (/FPc87), calls to emulator (/FPc), or alternate math (/FPa) option.

If you choose to use in-line instructions for your precompiled object files, you cannot link with an alternate math library (*x*LIBFA.LIB). However, in-line instructions achieve the best performance from your programs on machines that have an 80x87 coprocessor installed.

If you choose to use calls, your programs are slower, but at link time you can switch to any standard C run-time library that supports the memory model you have chosen.

Compatibility between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link two or more source files to produce an executable program file, you must ensure that floating-point operations are handled consistently and that the correct libraries are specified when you link your program.

If you are building libraries of C routines that contain floating-point operations, the calls to emulator option (/FPc) provides the most flexibility.

The examples that follow illustrate how you can link your program with a library other than the default. The floating-point option and the substitute library are compatible.

The example below compiles the program CALC.c with the medium model option (/AM). Because no floating-point option is specified, the default in-line emulator option (/FPi) is used. The in-line emulator option generates 80x87 instructions.

```
CL
  [Args]  /AM CALC.c
```

The example below compiles CALC.c using the small (default) memory model and the alternate math option (/FPa):

```
CL
  [Args]  /FPa CALC.c
```

The example below compiles CALC.c with the calls to math coprocessor option (/FPc87).

```
CL
  [Args]  /FPc87 CALC.c
```

Using the NO87 Environment Variable

Programs compiled using either the calls to emulator (/FPc) or the in-line emulator (/FPi) option automatically use an 80x87 coprocessor at run time if one is installed. You can override this and force the use of the software emulator by setting an environment variable named NO87.

If NO87 is set to any value when the program is executed, use of the coprocessor is suppressed. The value of the NO87 setting is printed on the standard output as a message. The message is printed only if a coprocessor is present and suppressed; if no coprocessor is present, no message appears. If you do not want a message to be printed, set NO87 equal to one or more spaces. A blank string for NO87 causes a blank line to be printed.

Note that only the presence or absence of the NO87 definition is important in suppressing use of the coprocessor. The actual value of the NO87 setting is used only for printing the message.

The NO87 variable takes effect with any program linked with the emulator library EM.Lib. It has no effect on programs linked with the math coprocessor library 87.LIB, or programs linked with alternate math libraries (*x*LIBFA.LIB).

When a program that uses an emulator library is executed and an 80x87 coprocessor is present, the example below causes the message Use of coprocessor suppressed to appear:

```
SET ENVIRONMENT
      NO87=Use of coprocessor suppressed
```

The syntax below sets the NO87 variable to the space character. Use of the coprocessor is still suppressed, but no message appears.

```
SET ENVIRONMENT
      NO87=spacebar
```

This Page is Blank

Do Not Print

Section 10
Managing Development Projects with NMAKE

The CTOS Microsoft Program-Maintenance Utility (NMAKE) is a sophisticated command processor that simplifies project management. NMAKE simplifies management by allowing you to list information and commands in a file called a description file, which NMAKE reads and executes. You can enter all build information in description files using NMAKE syntax.

Generally, most programmers use NMAKE to build libraries, compile source files, and link object modules. Most often, you will start with a small description file, and then add commands, paths, file names, and other information as your project develops. NMAKE can also execute any command, allowing you to make backups, move files, and do other project-management tasks.

The advantage of using NMAKE over batch files is that NMAKE rebuilds out-of-date project files only: you do not spend time rebuilding up-to-date files. For example, if you update source code in one file from a list of thirty files, NMAKE rebuilds the files dependent on this code only. It does not rebuild the complete application from scratch. NMAKE also has many advanced features that help you manage complex projects, such as macros and a language syntax that provides precise control over the build process.

This section provides complete documentation for NMAKE. Information about NMAKE is also available in online Help. If you are familiar with MAKE, the predecessor of NMAKE, be sure to read Differences Between NMAKE and MAKE. Also, Appendix G, Sample NMAKE Description Files, contains several examples, from small to very large, that illustrate most NMAKE features.

This section covers these topics:

- How NMAKE Works
- NMAKE Command Form
- NMAKE Description Files
- Command Line Options
- In-Line Files
- NMAKE Operations Sequence
- Differences between NMAKE and MAKE

How NMAKE Works

NMAKE compares times and dates of files called targets and dependents. Normally, targets are files to create, such as .Run files. Dependents are files used to create targets, such as C source files.

When executed, NMAKE first reads a description file. The description file consists of one or more blocks of text that you supply. Each block typically lists a target, the target's dependents, and the command that builds the target. NMAKE then compares dates and times between targets and dependents. If a dependent was changed later than the target, NMAKE executes the command in the block to update the target.

NMAKE Command Form

The NMAKE command form uses the following arguments and syntax:

```
NMAKE
 [Args] [options] [macros] [targets] [descriptfile]
```

All command-line fields are optional. If you do not supply arguments, NMAKE looks for MakeFile, a default description file, and follows other defaults described in this subsection.

The *options* field lists NMAKE options are described under the heading Command-Line Options. For a list of all options and Environment variables, refer to Apppendix B.

The *macros* field lists macro definitions, which allow you to globally replace text in the description file. For more information, refer to the subhead Macros under the heading NMAKE Description Files. For the most part, macros work on the command line as they do in description files. However, blank spaces and tabs are handled differently.

The *targets* field lists targets to build. If you do not list targets, NMAKE builds the first target in the description file only. This is the only feature of targets, and you will find no further description in this section. (This feature departs from MAKE's behavior MAKE, NMAKE's predecessor. For details, refer to the heading Differences between NMAKE and MAKE.

The *descriptfile* field specifies a description file. If this field is absent, NMAKE defaults to the current directory and looks for MakeFile. In addition, with option /F you can specify any description file. For details, refer to the heading Command-Line Options. For all practical purposes, the description file is the all-important feature of NMAKE, and most of this section is devoted to describing how to make and use one.

Here is typical NMAKE command:

```
NMAKE
 [Args] /S 'program=sample' sort.run search.run
```

In this example, the command supplies four arguments: an option (/S), a macro definition ('program=sample'), and two targets (sort.run, search.run).

Because the command does not specify a description file, NMAKE looks for MakeFile, the default description file. Option /S suppresses command display during execution. The macro replaces every instance of program with sample in the description file. And the target specifications tell NMAKE to update targets Sort.Run and Search.Run.

NMAKE Description Files

NMAKE always reads a description file, either the default MakeFile, or one that you specify on the command line. If NMAKE cannot find a description file, and no targets are specified on the command line that can be built with default inference rules, it returns an error message.

Description files can contain these elements:

- Description Blocks
- Comments
- Macros
- Inference Rules
- Directives
- Pseudotargets

Description Blocks

Description blocks form the heart of description files. Figure 10-1 illustrates the three parts of a typical NMAKE description block: targets, dependents, and commands. The line that lists targets and dependents is called the dependency line. The line that lists command is called the command line. (Multiple lines of text can also appear as the dependency or command line.)

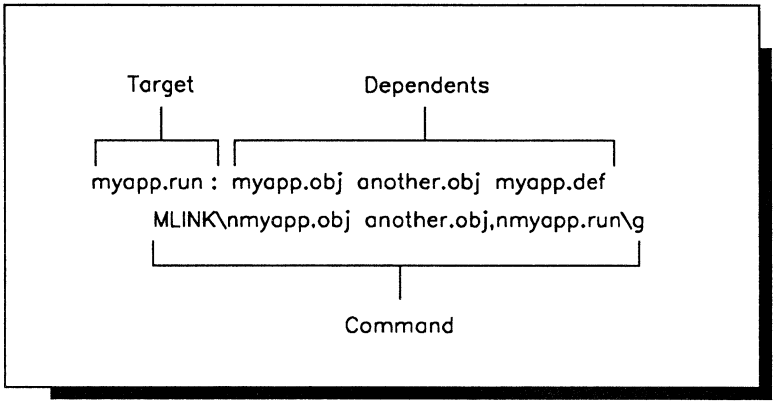


Figure 10-1. Typical Description Block

The target part lists one or more files to build. The example in Figure 10-1 tells NMAKE to build a single target, myapp.run. Although single targets are common, you can also list multiple targets, but you must separate target names with spaces.

Normally, the target is a file, but it can also be a pseudotarget, a name that allows you to build groups of files, or execute a group of commands. For details, refer to the subheading Pseudotargets.

The dependents part lists one or more files that NMAKE uses to create the target. A colon separates targets from dependents. The example in Figure 10-1 lists three dependents:

```
myapp.run : myapp.Obj another.Obj myapp.def
```

The example tells NMAKE to build the target myapp.run whenever myapp.Obj, another.Obj, or myapp.def has changed more recently than myapp.run.

If any dependents are listed as targets in other description blocks, NMAKE builds those files first, before building the specified target. As a result, NMAKE evaluates a dependency tree for the entire description file, rebuilding dependent files, if necessary, to update the specified target.

The dependent list can also include a list of directories where NMAKE searches for dependents. The directory list is enclosed in curly braces ({ }), and precedes the dependent list. NMAKE first searches the current directory, then searches the listed directories.

For example, consider the line:

```
forward.run : {[d1]<alpha>;<proj>}pass.Obj
```

Here, the target, forward.run, has one dependent: pass.Obj. Also notice that the directory list specifies two directories:

```
{[d1]<alpha>;<proj>}
```

Therefore, NMAKE first searches for pass.Obj in the current directory. If not found, NMAKE next searches [d1]<alpha>, then searches <proj>. If NMAKE cannot find a dependent in either the current or listed directories, it looks for an inference rule that describes how to create the dependent.

The command part of the description block lists the command(s) NMAKE should use to build the target. Any command executed from the command line in valid. The example in Figure 10-1 tells NMAKE to build myapp.run using the MLink command:

```
MLINK\nmyapp.Obj another.Obj,nmyapp.run\g
```

Notice that in Figure 10-1, this line is indented. To distinguish between the dependency line and command line, NMAKE uses indentation. Thus, if the command appears on a separate line, as here, it must be indented at least one space or tab. Further, do not indent the dependency line (space or tab).

While many targets are built with single commands, you can place multiple commands after the dependency line. A long command can span several lines as long as each line ends with a backslash (\ ).

You can also place the command at the end of the dependency line, but you must separate the command from the rightmost dependent with a semicolon.

With this background in mind, you can also incorporate the following advanced features into description blocks:

- Escape Sequences
- Wildcards
- Command Modifiers
- Control Characters as Literals
- Targets in Multiple Description Blocks

Escape Sequences

Within description files, CTOS supports three escape sequences: \f, \g, and \n.

NMAKE interprets the characters this way:

\f End-of-file character.

\g GO character

\n Newline character

For example, to signal a move from one argument line to the next in a command form, use \n.

Wildcards

When specifying target and dependent file names, you can use the wildcard characters asterisk (*) and question mark (?). When reading description files, NMAKE expands wildcards in target names. When building targets, NMAKE expands wildcards in dependent names. For example, the following description block compiles all source files with the .c extension:

```
bondo.run : *.
 CL\n*.c\g v
```

Command Modifiers

Command modifiers, special characters that precede commands, control the commands listed in description blocks. You can use one or both modifiers for a single command. Table 5-1 describes the two NMAKE command modifiers.

Table 10-1. Command Modifiers

Character	Action

At sign (@)	Stops NMAKE from displaying commands during execution. The following example suppresses the DELETE command from appearing on the screen:
	
	`sort.Obj : sort.c`
	`  @DELETE\n sort.Obj\g`

Exclamation point (!)	Executes the command for each dependent file if the command uses the predefined macros $? or $**. $? refers to all out-of-date dependent files with respect to targets. $** refers to all dependent files in the description block (see Macros). For example:
	
	`print:hop.Asm skip.bas jump.c`
	`  !print\n$**\n[spl]\g`
	
	generates the following commands:
	
	`print\nhop.Asm\n[spl]\g`
	`print\nskip.bas\n[spl]\g`
	`print\njump.c\n[spl]\g`

Using Control Characters as Literals

Occasionally, you may need to list file names that contain special characters that NMAKE uses as control characters. The control characters are:

```
# () $ ^ \  { } ! @ -
```

To use control characters as literal characters, precede control characters with a caret (^).

For example, suppose a macro ends with a backslash:

```
runpath=[Sys]<Sys>run\
```

This line intends to define a macro named runpath with value [Sys]<Sys>run\ . However, the backslash causes unexpected results because NMAKE considers it the line-continuation character. So the line defines the macro runpath as [Sys]<Sys>run, but then appends what appears on the next line.

To solve this problem, place a caret in front of the backslash:

```
runpath=[Sys]<Sys>run^\
```

You can also use a caret to place a literal newline character in a description file. This feature can be useful in macro definitions:

```
XYZ=abc^

def
```

NMAKE interprets this example as if you assigned the C-style string abc \n def to the XYZ macro. This effect differs from using the backslash (\ ) to continue a line. A newline character that follows a backslash is replaced with a space.

NMAKE ignores carets that precede noncontrol characters. For example, NMAKE interprets:

```
ign^ore : these ca^rets
```

as:

```
ignore : these carets
```

Further, NMAKE treats carets that appear in quotation marks as literal characters.

Listing Targets in Multiple Description Blocks

To specify multiple description blocks for the same target, place two colons (::) after the target. This feature is useful for building complex targets, such as libraries that contain components created with different commands.

For example, consider these blocks:

```
target.Lib :: a.Asm b.Asm c.Asm
      MASM\na.Asm b.Asm c.Asm\g
      MLIB\ntarget -+a.Obj -+b.Obj -+c.Obj;\g
target.Lib :: d.c e.c
      CL\nd.c e.c\g
      MLIB\ntarget -+d.Obj -+e.Obj;\g
```

Here, both description blocks update library TARGET.LIB. If any of the assembly-language files change later than the library, NMAKE executes the commands in the first block, assembles the source files, and updates the library. Similarly, if any of the C-language files change, NMAKE executes the commands in the second block, compiles the C files, and updates the library.

If you use a single colon in multiple blocks, NMAKE returns an error. However, single colons are legal for commands listed in one-line blocks. In this situation, dependency lines are cumulative.

For example:

```
target: jump.bas
target: up.c
     copy\n[KBD]\n[VID]\gBuilding target\f
```

equates to:

```
target: jump.bas up.c
     copy\n[KBD]\n[VID]\gBuilding target\f
```

Comments

To place comments in a description file, precede text with a number sign (#). A comment extends to the end of the line on which it appears.

This example contains two comments:

```
# This comment gets its own line
  ducks.run : huey.Obj dewy.Obj # Comment to end a line
  link\nhuey.Obj dewy.Obj\nducks.run\g
```

Note: *Command lines cannot contain comments, and comments cannot be nested.*

Macros

Macros offer convenient ways to replace strings in description files with other strings. Each time you run NMAKE, text is replaced automatically. Thus, macros have many uses, but are especially useful when used to:

- Create a standard description file for several projects

  Create a macro to represent file names used in commands. When run, NMAKE substitutes file names for the macro. This allows you to use the macro on different projects, because you can now change file names throughout description files by changing a single macro.

- To control the options that NMAKE passes to the compiler or linker.

  Create a macro to specify the options you use most often when compiling or linking.

You can define your own macros or use predefined macros, use macros in description files and on the command line.

The subheadings that follow describe these macro topics:

- User-Defined Macros
- Invoking Macros
- Predefined Macros
- Substitution Within Macros
- Inherited Macros
- Precedence among Macro Definitions

User-Defined Macros

The following syntax shows a macro definition:

```
macroname = string
```

macroname can be any combination of letters, digits, and the underscore ( ) character. Macro names are also case sensitive. For example, NMAKE interprets MyMacro and MYMACRO differently.

string can be any string, including a null string.

For example:

```
command = MLINK
```

defines a macro named command and assigns it the string MLINK.

You can define macros in the description file or on the command line. However, the rules for handling spaces are different.

In description files, you must define each macro on separate lines, but the line must not begin with spaces or tabs. You can, however, embed spaces in *string,* and place spaces on either side of the equal sign. Further, if you enclose *string* in quotation marks, they become part of the string.

On the command line, different rules apply for handling spaces. Here, you cannot embed spaces in *string* unless you enclose *string* in quotation marks. Also, you cannot place spaces around the equal sign. You can, however, enclose the entire macro definition, *macroname* and *string,* in single quotation marks, as shown in this example:

```
NMAKE
 [Args]   'program=sample'
```

Here, the macro program is assigned the value sample.

Also, once you define a macro, you can undefine it with the directive !UNDEF.

Invoking Macros

To use macros in description files, enclose macro names in parentheses preceded by a dollar sign ($). (If *macroname* is one-character only, parentheses are optional.)

For example, the macro defined with:

```
command=MLINK
```

becomes active when this syntax occurs in description files:

```
$ (command)
```

Thus, when run, NMAKE replaces every occurrence of command with MLINK.

The following description file defines and uses three macros:

```
program = sample
c = MLINK
optional =
$ (program).run : $(program).Obj
    $c\n$(program).Obj$(optional)\n$(program).run\g
```

NMAKE interprets this description block as:

```
sample.run : sample.Obj
  MLINK\nsample.Obj\nsample.run\g
```

Here, NMAKE replaces every occurrence of $(program) with sample, every instance of $c with MLINK, and every instance of $(optional) with a null string.

Because c is one character only, parentheses are optional.

NMAKE treats undefined macros as null strings. -----

To use the dollar sign ($) as a literal character, use two signs ($$), or precede it with a caret (^$).

Predefined Macros

NMAKE provides several predefined macros, which represent various file names and commands. Predefined macros are useful in their own right, and also affect inference rules. Table 10-2 lists NMAKE predefined macros.

Table 10-2 . Predefined Macros

Macro	Meaning
$@	Current target's full name.
$*	Current target's base name (full name minus the file extension).
$**	Dependents of the current target.
$?	Dependents that are out-of-date with respect to the current target.
$$@	Target that NMAKE is currently evaluating. You can use this macro to specify a dependent only.
$<	Dependent file that is out-of-date with respect to the current target (evaluated only for inference rules).
$(CC)	Command to invoke the C compiler. By default, is predefined as CC=cl, which invokes the optimizing compiler.

continued

Table 10-2 . Predefined Macros (cont.)

Macro	Meaning
$(AS)	Command that invokes the CTOS Microsoft Macro Assembler. NMAKE defines this macro as AS = masm.

$(MAKE)	Command name that invokes NMAKE utility. This macro invokes NMAKE recursively, and builds different versions of a program. The following description file invokes NMAKE recursively to build targets in the VERS1 and VERS2 directories.
	
	`all :vers1 vers2`
	`vers1 :`
	`  path\n\nvers1\g`
	`  $(MAKE)\g`
	`  path . .`
	`vers2 :`
	`  path\n\nvers2\g`
	`  &(MAKE)\g`
	`  path . .`
	
	In this example, the system starts in the VERS1 directory, then invokes NMAKE recursively, causing NMAKE to process the file MakeFile in that directory. When finished, the system changes to the VERS2 directory and invokes NMAKE again, processing MakeFile in that directory.

$(MAKEFLAGS)	NMAKE options currently in effect. If you invoke NMAKE recursively, use the command: $/(MAKEFLAGS). You cannot redefine this macro.
$(MAKEDIR)	Directory from which NMAKE is invoked.

Like user-defined macro names, predefined macro names are also case sensitive. Thus, NMAKE interprets CC and cc differently.

To modify any of the first six macros in Table 10-2, append one of these characters:

D Specifies the directory part of the file name only.

F Specifies the file name only.

B Specify the base name only.

S Specifies the complete file name without the extension.

If you add one of these characters, enclose the macro name in parentheses. (The macros $$@ and $** are the only exceptions to the rule that macro names longer than one character must be enclosed in parentheses.)

For example, assume that $@ has the value [Sys]<proj>sort.Obj. The following list shows the effect of combining special characters with $@:

Macro	Value
$(@D)	[Sys]<prog>
$(@F)	sort.Obj
$(@B)	sort
$(@R)	[Sys]<proj>sort

In the code that follows, the macro $? represents names of all dependents changed later than the target. The exclamation point executes the MLIB command once for each dependent listed. As a result, MLIB executes up to three times, each time replacing a module with a newer version.

```
trig.Lib : sin.Obj cos.Obj arctan.Obj

      !MLIB\ntrig.Lib -+$?;\g
```

In the next example, NMAKE updates a group of include files:

```
# Include files depend on versions in current directory
DIR=[Sys]<include>
$(DIR)globals.h : globals.h
     COPY\nglobals.hbsn$@\g
$(DIR)types.h : types.h
     COPY\ntypes.h\n$@\g
 $(DIR)macros.h : macros.h
     COPY\nmacros.h $@\g
```

Each of these files, globals.h, types.h, and macros.h, in directory [Sys]<include>, depends on its counterpart in the current directory. If an include file is out-of-date, NMAKE replaces it with the file of the same name from the current directory.

Substitution within Macros

Just as macros allow you to substitute text in description files, you can also substitute text within macros, using the syntax:

```
(macroname:string1 = string2)
```

Here, in the macro *macroname, string1* is replaced by *string2*. Spaces or tabs between *macroname* and the colon are not allowed. Also, spaces between the colon and *string1* become part of *string1*. And if string2 is null, NMAKE deletes all occurrences of *string1* from the macroname macro.

The following description file illustrates macro substitution:

```
SRCS = prog.c sub1.c sub2.c
  prog.run : $(SRCS: c =.Obj)
       MLINK\n$**\n$@\g
```

The predefined macro $** stands for the names of all the dependent files. Also, if you invoke the example file with a command line that specifies the target, NMAKE executes the following command:

```
MLINK\nprog.Obj sub1.Obj sub2.Obj\nprog.run\g
```

Macro substitution does not alter the definition of the SRCS macro. It simply replaces the listed characters. When NMAKE builds the target prog.run, it gets the definition for the predefined macro $** (the dependent list) from the dependency line, which specifies the macro substitution in SRCS.

You can also perform substitution in the following predefined macros: $@, $*, $**, $?, and $\<. The substitution principle is the same as for other macros.

For example, the command in the following description block performs substitution within a predefined macro:

```
target.abc : depend.xyz
 command\n:targ=blank)\g
```

If dependent dpnd.xyz is out-of-date relative to target trgt.abc, then NMAKE executes this command:

```
command\nblanket.abc\g
```

This example uses the predefined macro $@, which equates to the full name of the current target (trgt.abc). It substitutes blank for targ in the target, resulting in blanket.abc.

Note, however, that the usual dollar sign does not appear in front of the predefined macro.

The example uses:

```
$(@:targ=blank)
```

instead of:

```
$($@:targ=blank)
```

to substitute within the predefined macro $@.

Inherited Macros

When executing, NMAKE creates macros equivalent to every current environment variable. These macros are called inherited because their names match the corresponding environment variables. (The inherited macro is all uppercase, however, even if the corresponding environment variable is not.)

You can use inherited macros like other macros, and also redefine them. The following example redefines the inherited macro PATH:

```
PATH = [Sys]<tools>

sample.Obj : sample.c
    CL\nsample.c\g
```

When NMAKE executes CL, no matter what value PATH started with, CL works in path [Sys]<tools>. However, redefining the inherited macro does not affect the original environment variable; when NMAKE terminates, PATH retains its original value.

Option /E defeats macro inheritance. With this option, NMAKE ignores any attempt to redefine a macro derived from an environment variable.

Precedence Among Macro Definitions

If you define the same macro in more than one place, NMAKE uses the macro with the highest precedence. The following list shows precedence from highest to lowest:

1. Macros defined on the command line
2. Macros defined in a description file or include file
3. Inherited macros
4. Predefined macros such as CC and AS

Option /E defeats any attempt to redefine inherited macros. If you run NMAKE with this option, macros inherited from environment variables over-ride any same-named macros in the description file.

Inference Rules

To create files without specified extensions, NMAKE applies templates called inference rules. For example, when handling a description block without commands, NMAKE applies an inference rule that describes how to create the target from the dependent files, using two specified extensions. Similarly, if a dependent file does not exist, NMAKE applies an inference rule that describes how to create the missing dependent from another file with the same base name.

In short, inference rules provide a convenient shorthand for common operations. For instance, you can use an inference rule to avoid repeating the same command in several description blocks.

The subheadings that follow describe these inference-rule topics:

- User-Defined Inference Rules
- Extension Search Paths
- Predefined Inference Rules
- Precedence Among Inference Rules

User-Defined Inference Rules

You can define inference rules in the description file. An inference-rule definition lists two file extensions, and one or more commands. For instance, the following rule tells NMAKE how to build an .OBJ file using a .c file:

```
.c.OBJ:
    CL\n$<\g
```

The first line lists two extensions. The second extension (.OBJ) specifies the type of the desired file and the first (.c) specifies the desired file's dependent. The second line lists the command used to build the desired file. Here, the predefined macro $< represents a dependent out-of-date relative to the target.

NMAKE could apply the above inference rule to the following description block:

```
sample.Obj :
```

This description block lists a target only: SAMPLE.OBJ. Both the dependent and the command are missing. However, given the target's base name and extension, plus the above inference rule, NMAKE has enough information to build the target. NMAKE first looks for a .c file with the same base name as the target. If SAMPLE.c exists, NMAKE compares its date to that of SAMPLE.OBJ (the comparison is triggered by the predefined macro $<). If SAMPLE.c has changed more recently, NMAKE compiles it using the CL command listed in the inference rule:

```
CL\nsample.c\g
```

However, NMAKE applies an inference rule only if the base name of the file to create matches the base name of an existing file. Thus, inference rules are useful only when a one-to-one correspondence exists between the desired file and its dependent. For example you cannot define an inference rule that replaces several modules in a library.

Extension Search Paths

If an inference rule does not specify a search path, as shown in this example,

```
.c.Obj:
 commands. . .
```

NMAKE looks for files in the current directory. However, you can specify a single path for each extension, using this syntax:

```
{frompath}.fromext {topath}.toext:
      commands
```

Here, NMAKE searches in the frompath directory for files with extension fromext, and uses commands to create files with extension toext in directory topath.

The following examples specify search paths. The first example is simple; the second example shows some sophistication:

```
{[Sys]<Source>}.c{[Sys]<Build>}.Obj:
commands...

SRCDIR=[Sys]<Source>
BLDDIR=[Sys]<Build>
{$(SRCDIR)}.c{$(BLDDIR)}.Obj:
commands...
```

Predefined Inference Rules

NMAKE provides predefined inference rules to perform these common development tasks:

- Create an .Obj file by compiling a .c file
- Create an .Obj file by assembling an .Asm file

Table 10-3 lists the predefined inference rules.

Table 10-3. Predefined Inference Rules

Inference Rule	Command	Default Action
.c.Obj	$(CC)\n/c$*.c\ g	cl\ n/c $*.c\ g
.Asm.Obj	$(AS)\n $*;\g	masm\ n$*;\ g

Precedence among Inference Rules

If the same inference rule is defined in more than one place, NMAKE uses the rule with the highest precedence. The precedence from high to low is:

1. User-defined Inference rules in description files
2. Predefined inference rules

NMAKE uses a predefined inference rule only if no user-defined inference rule exists for the desired operation.

Directives

Using directives, you can write description files that act in a manner similar to batch files. Directives can execute commands conditionally, display error messages, include other files, and turn on or off certain options.

Directives begin with an exclamation point (!), which must appear at the beginning of the line. Spaces are legal between the exclamation point and directive keyword. Table 10-4 lists directives, and describes how they act.

Table 10-4. Directives

Directive	Description
!CMDSWITCHES {+\v-}opt.	Turns on or off one of four NMAKE options: /D, /I, /N, and /S. If unspecified, NMAKE resets the options to initial values when NMAKE started. To turn on an option, precede it with a plus sign (+); to turn off an option, precede it with a minus sign (-). Also, using this keyword updates macro MAKEFLAGS.
!ELSE	Executes statements between keywords !ELSE and !ENDIF, if statements preceding !ELSE were not executed.
!ENDIF	Marks the end of !IF, !IFDEF, or !IFNDEF statement blocks.
!ERROR *text*	Prints *text*, then stops execution.
!IF *constantexpression*	Executes statements between !IF and the next !ELSE or !ENDIF, if *constantexpression* evaluates to a nonzero value only.

continued

Table 10-4. Directives (cont.)

Directive	Description
!IFDEF *macroname*	Executes statements between !IFDEF and the next !ELSE or !ENDIF, if *macroname* is defined only. NMAKE considers null-value macros as defined.
!IFNDEF *macroname*	Executes statements between !IFNDEF and the next !ELSE or !ENDIF, if *macroname* is not defined only.
!INCLUDE *filename*	Reads and evaluates file *filename* before continuing with the current description file. If *filename* is enclosed with angle brackets (<>), NMAKE searches for the file in the directories specified by the INCLUDE macro. Otherwise, it searches the current directory only. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.
!UNDEF *macroname*	Marks *macroname* as undefined in NMAKE's symbol table.

The *constantexpression* used with !IF can consist of integer constants, string constants, or program invocations. Integer constants can use C unary operators (-) (for numerical negation), one's complement (~), and logical negation (!). Integer constants can also use any C binary operator listed in Table 10-5.

Table 10-5. Directive Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^^	Bitwise XOR
&&	Logical AND
||	Logical OR
	Left shift
>>	Right shift
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

You should remember these rules. To group expressions, use parentheses. NMAKE treats numbers as decimal unless they start with 0 (octal) or 0x (hexadecimal). To compare two strings for equality, use the equality (==) operator. To compare two strings for inequality, use the inequality (!=) operator. Enclose strings with quotes. Enclose program invocations in square brackets ([ ]).

The following example illustrates how to use directives:

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.run:winner.Obj
! IFDEF debug
!  IF "$(debug)"=="y"
       MLINK\nwinner.Obj,nwinner.run\g
   ELSE
       MLINK\nwinner.Obj,nwinner.dbg\g
!  ENDIF
! ELSE
!  ERROR Macro named debug is not defined.
! ENDIF
```

Here, !INCLUDE causes NMAKE to insert file infrules.txt into the description file. !CMDSWITCHES turns on option /D, which displays file dates as they are checked. If winner.run is later than winner.Obj, !IFDEF checks if macro debug is defined. If defined, !IF checks inspects whether it is set "y." If set to "y," the linker executes with runfile winner.run; if macro debug is not "y,"the linker executes with file winner.dbg. If the debug macro is not defined, the !ERROR directive prints the message and NMAKE stops.

Pseudotargets

Pseudotargets are useful for building groups of files, and for executing groups of commands.

Pseudotargets are similar to targets, but they are not files. Pseudotargets are names that serve as handles for building groups of files, or for executing groups of commands.

For example, here is a pseudotarget called ENVIRON:

```
ENVIRON:
      Set environment\nCL=/AL\g
```

When NMAKE evaluates pseudotargets, it always considers dependents as out-of-date. In the above example, NMAKE copies each dependent file to the specified drive and directory.

Like macro names, pseudotarget names are case sensitive. Predefined pseudotarget names are all uppercase.

Table 10-6 lists predefined pseudotargets, which provide special rules in description files. You can use these names on command lines or in description files. However, you need not specify them as targets; NMAKE uses the rules they define no matter where they appear.

Table 10-6. Pseudotargets

Pseudotarget	Action
.IGNORE:	Ignores exit codes returned by programs called from description files. Same effect as invoking NMAKE with option /I.
.SILENT:	Suppresses line display as they are executed. Same effect as invoking NMAKE with option /S.
.SUFFIXES:*list*	Lists file suffixes for NMAKE to try when building target files missing specified dependent. NMAKE uses this list along with inference rules. When NMAKE finds a target without dependents, it searches the current directory for a file with a base name matching the target, and matching a suffix from the list. If NMAKE finds such a file, and if an inference rule applies to the file, NMAKE treats the file as a dependent of the target. Further, suffix order in the list defines the order NMAKE follows to search for the file. The list is predefined as follows: .SUFFIXES: .Obj .run .c .Asm To add suffixes, specify .SUFFIXES:, then list the new suffixes. To clear the list, specify .SUFFIXES:

Command Line Options

NMAKE accepts a number of options, which are listed in Table 10-7. You can specify options in uppercase or lowercase, and use either a slash or dash. For example, -A, /A, -a, and /a all represent the same option.

Table 10-7. NMAKE Options

Option	Action
/A	Builds all requested targets, even if not out-of-date.
/C	Suppresses nonfatal error or warning messages, and the NMAKE logo.
/D	Displays modification dates of all files.
/E	Causes environment variables to override macro definitions in description files.
/F *filename*	Specifies *filename* as the name of the description file.
/Y	Forces NMAKE to use deferred rather than inline processing.
/HELP or /?	Displays a brief summary of NMAKE syntax, and exits to the operating system.
/I	Ignores return codes from commands listed in the description file. NMAKE processes the whole description file even if errors occur.
/L	Suppresses the NMAKE logo.
/N	Displays but does not execute the description file's commands. This option is useful for debugging description files and checking which targets are out-of-date.
/P	Displays all macro definitions and target descriptions on the standard output device.

continued

Table 10-7. NMAKE Options (cont.)

Option	Action
/Q	Returns zero if the target is up-to-date, and nonzero if not.
/S	Suppresses the display of commands listed in the description file.
/T	Changes modification dates for out-of-date target files to the current date.
/X *filename*	Sends all error output to *filename,* which can be a file or a device. To send error output to the standard output device, enter a dash (-) instead of a file name.
/Y	Forces NMAKE execution to use CTOS submit methodology instead of spawning commands.

The following command specifies two NMAKE options:

```
NMAKE
    [Args]  /f sample.mak /c targ1 targ2
```

Here, option /f tells NMAKE to read the description file SAMPLE.MAK. Option /c tells NMAKE not to display nonfatal error messages and warnings.

The following command lists two targets to update.

```
NMAKE
    [Args]  /D /N targ1 targ1.mak
```

Here, NMAKE updates target targ1. If the current directory does not contain a file named MAKEFILE, NMAKE reads the file TARG1.MAK as the description file. Option /D displays modification dates of each file; option /N displays commands without executing them.

In-Line Files

NMAKE can write in-line files, which can contain any text. One use of this file type is as a response file for another utility such as MLIB. (Response files supply programs with long argument lists that exceed the length of the command line.)

To create an in-line file, use this syntax:

```
target : dependents
   command << [filename]
  inlinetext
  <<[KEEP | NOKEEP]
```

All text between the two sets of double angle brackets (<<) apears in the in-line file. The filename is optional. If you do not supply filename, NMAKE gives the in-line file a unique name: either [scr]<$>\g, or if the TMP environment variable is defined, in the TMP directory.

The in-line file can be temporary or permanent. To retain the file, specify KEEP. Otherwise, specify NOKEEP, which is also the default.

The following example creates an MLIB response file named LIB.LRF:

```
math.Lib : add.Obj sub.Obj mul.Obj div.Obj
    MLIB\n~<<lib.lrf
 math.Lib
 -+add.Obj-+sub.Obj-+mul.Obj-+div.Obj
 listing;
 <<KEEP
    \g
```

The resulting response file tells MLIB which library to use, which commands to execute, and which listing file to produce:

```
math.Lib

 -+add.Obj-+sub.Obj-+mul.Obj-+div.Obj
 listing
```

Also, the in-line file specification can create more than one in-line file.

For example:

```
target.abc : depend.xyz
    append\n <<file1 <<file2
  I am the contents of file1.
  <<KEEP
  I am the contents of file2.
  <<KEEP
     \nfile3\g
```

Here, NMAKE creates two in-line files named FILE1 and FILE2; then executes the command:

```
append\nfile1 file2 file3\g
```

The KEEP keywords tell NMAKE not to delete FILE1 and FILE2 when finished.

NMAKE Operations Sequence

To write complex description files, you need to understand how NMAKE interprets description files.

When run from the command line, NMAKE first looks for the description file, following these steps:

1. NMAKE looks for option /F. If found, it uses the file listed as the description file.
2. If option /F is not specified, NMAKE looks in the current directory for a file named MakeFile. If found, it becomes the description file.
3. If MakeFile is not in the current directory, NMAKE parses the command line for the first string that is not an option or macro definition, and treats this string as a file name.

   If the file-name extension does not appear in the .SUFFIXES list, NMAKE uses the file as the description file. If the extension appears in the .SUFFIXES list, NMAKE tries additional strings until it finds a suitable file. For details on .SUFFIXES, refer to Pseudotargets in this seciton.
4. If NMAKE cannot find a description file, it returns an error.

NMAKE stops searching for a description file as soon as it finds one, even if other potential description files exist. For example, if you specify /F, NMAKE uses the specified file even if MakeFile exists in the current directory. Similarly, if NMAKE uses MakeFile, any description file listed in the command line becomes a target.

Next, NMAKE updates every target listed on the command line. If none appear, NMAKE updates only the first target in the description file.

NMAKE then applies macro definitions and inference rules in the following order, from highest to lowest priority:

1. Macros defined on the command line
2. Macros defined in a description file or include file
3. Inherited macros
4. Predefined macros such as CC and AS

Definitions in later steps take precedence over definitions in earlier steps. Option /E, however, causes inherited macros to override macros defined on the command line.

Now NMAKE updates each target in the order listed in the description file. It compares date and time of dependents with targets, and executes commands that update targets. If you specify option /A, or if the target is a pseudotarget, NMAKE updates the target even if its dependents are not out-of-date.

If the target has no explicit dependents, NMAKE searches the current directory for one or more files with extensions that appear in the .SUFFIXES list. If found, NMAKE treats the files as dependents, and updates the target according to the commands.

If no commands are listed to update the target, or if NMAKE cannot find the dependents, NMAKE builds the target according to inference rules. By default, it tries to build .RUN files from .OBJ files; then tries to build .Obj files from .c and .Asm sources. In practice, this means you should specify .OBJ files as dependents, because NMAKE compiles your source files when it cannot find the .Obj files.

Differences between NMAKE and MAKE

As its name implies, NMAKE is a new utility that replaces the older Microsoft MAKE program. NMAKE differs from MAKE in the following ways:

- NMAKE does not evaluate targets sequentially. Instead, NMAKE updates the targets you specify when you invoke it, regardless of their positions in the description file. If no targets are specified, NMAKE updates only the first target in the file.
- NMAKE accepts command-line arguments from a file.
- NMAKE provides more command-line options.
- NMAKE provides more predefined macros.
- NMAKE permits substitutions within macros.
- NMAKE supports directives placed in the description file.
- NMAKE allows you to specify include files in the description file.

The first item in the list deserves special emphasis. While MAKE normally builds every target, working from beginning to end of the description file, NMAKE expects you to specify targets on the command line. If you do not, NMAKE builds only the first target in the description file.

The difference is clear if you run NMAKE using a typical MAKE description file, which lists a series of subordinate targets followed by a higher-level target that depends on the subordinates:

```
app.Obj : app.c
   CL \n/c /G2sw /W3 app.c\g

 app.run : app.Obj
   MLINK \napp\g
```

MAKE builds both targets (app.Obj and app.run), but NMAKE builds only the first target (app.Obj).

Because of these performance differences, you may want to convert MAKE files to NMAKE files. MAKE description files are easy to convert. A simple method is to create a new description block at the beginning of the file. Give this block a pseudotarget named ALL and list the top-level target as a dependent of ALL. To build ALL, NMAKE must update every target upon which the target of ALL depends:

```
ALL : app.run

 app.Obj : app.c
   CL \n/c /G2sw /W3 app.c\g

 app.run : app.Obj
   MLINK \napp\g
```

If the above file is named makefile, you can update the target app.run using the NMAKE command with no arguments, or:

```
NMAKE
   [Args] ALL
```

Note that it is not necessary to list app.Obj as a dependent of ALL. NMAKE builds a dependency tree for the entire description file, and builds whatever files are needed to update app.run. So if app.c is out-of-date with respect to app.Obj, NMAKE compiles app.c to create app.Obj, then links app.Obj to create app.run.

The same technique is suitable for description files with more than one top-level target. List all of the top-level targets as dependents of ALL:

```
ALL : app.run second.run another.run
```

The example updates the targets app.run, second.run, and another.run.

If the description file lists a single, top-level target, you can use an even simpler technique. Move the top-level block to the beginning of the file:

```
app.run : app.Obj
   MLINK \napp\g

 app.Obj : app.c
   CL \n/c /G2sw /W3 app.c\g
```

NMAKE updates the second target (app.Obj) whenever needed to keep the first target (app.run) current.

Section 11
Creating Help Files with HelpMake

The CTOS Microsoft Help File Creation Utility (HelpMake) allows you to create your own Help files for use with CTOS or Microsoft products. It also allows you to customize Help files supplied with CTOS Microsoft C and Microsoft language products. HelpMake translates Help text files into a database accessible from the QuickHelp utility (command), and from the Programmer's WorkBench (PWB).

This section describes how to create and modify Help files using HelpMake, and covers the following topics:

- Help Database Content and Format
- Using the HelpMake Command
- Using HelpMake Command Options
- Modifying a Help Database
- Help Text Conventions
- Using Help Database Formats

Help Database Content and Format

HelpMake creates a database from one or more input files that contain properly formatted information. This subsection provides an overview of how to format database contents, and outlines the three standard formats that HelpMake supports.

Help File Contents

Each Help text file starts with a topic and information about the topic, then lists another topic and more information, and so on. In HelpMake terminology, topics are called contexts; the information is called topic text.

The .context command introduces a context. For example, in the source Help file for the CTOS Microsoft C language (Clang.Hlp), this line introduces Help for the open function:

```
.context open
```

open is analagous to a global label: it is a link to the topic text. The **.context** command and other formatting elements are described under the heading Help Text Conventions.

Often, especially with library routines, the same information applies to more than one subject. For example, the string-to-number functions strtod, strtol, and stroul share the same topic text. The Help file lists all three function names as contexts for one block of topic text. The converse, however, is not true. You cannot specify different blocks of topic text in different places in the Help file using the same .context label.

Cross references make it possible to view information about related topics, including header files and code examples. The Help for the **open** function, for example, references the **access** function and the ASCII header file FCNTL.h. cross references can point to other contexts in the same Help database, to contexts in other Help databases, or to ASCII files outside the database.

Help files can have two kinds of cross references:

- Implicit
- Explicit, or hyperlinks

The word **open** is an implicit cross reference throughout CTOS Microsoft C Help. If you select the word **open** anywhere in QuickHelp, the Help system displays information on the **open function**. As illustrated above, the context for open begins with an ordinary **.context** command. As a result, anywhere that you select open, the Help system references this context.

A hyperlink is an explicit cross reference tied to a word or phrase at a specific location in the Help file. You create hyperlinks when you write the Help text. The hyperlink consists of a word or phrase followed by invisible text that gives the context or label of the hyperlink.

For example, to cause a particular instance of the word **formatting** to display Help on the printf function, you would create an explicit cross reference from the word **formatting** to the context printf. Elsewhere in the file, **formatting** has no special significance but, at that one position, it references the Help for printf. For more information, refer to the heading Hyperlinks.

Help text can also include formatting flags to control the appearance of the text on the screen. Using these flags, you can make certain words appear in various colors, reverse video, and so forth, depending on the application displaying Help and the graphics capabilities of the host computer.

Help File Formats

You can create Help files using three standard formats:

- QuickHelp format
- Minimally formatted ASCII
- Unformatted ASCII
- Rich Text Format (RTF)

In addition, you can reference unformatted ASCII files, such as include files, from within a Help database.

An entire Help system (such as the one supplied with CTOS Microsoft C) can use any combination of files formatted with different format types. With C, for example, the README.DOC information file is unformatted ASCII; the Help files for PWB, the C language, and run-time library are encoded in the QuickHelp format. The databases also cross reference the header (include) files, which are also ASCII files stored outside the database.

QuickHelp Format

QuickHelp format is the default. It is the format into which HelpMake decodes Help databases. You can use any text editor to create a QuickHelp-format Help text file. QuickHelp format also lends itself to a relatively automated translation from other document formats.

QuickHelp files can contain any kind of cross reference or formatting attribute like dot commands and formatting flags. Typically, you use QuickHelp format for any changes to a database that CTOS Microsoft C supplied.

Minimally Formatted ASCII

Minimally formatted ASCII files simply define contexts and their topic text. These files cannot contain screen-formatting commands or explicit cross references (implicit cross references are allowed). They are often used to display small Help files that do not require compression.

Unformatted ASCII

Unformatted ASCII files are exactly what their name implies: regular ASCII files with no special formatting commands, context definitions, or special information. An unformatted ASCII file does not become part of the Help database. Only its name is used as the object of a cross reference. The standard C header (include) files are unformatted ASCII files used for cross references by the Help system for the C run-time library. Unformatted ASCII files are also useful for storing program examples.

Rich Text Format

Many word processors support Rich Text Format (RTF). However, CTOS does not support RTF. You can create RTF Help text with any word processor that generates RTF output. You can also use any utility program that takes word processor output and produces an RTF file.

Use RTF when you want to transfer Help files from one application to another while retaining formatting information. You can format RTF files directly with the word processing program; you need not edit them to insert any special commands or tags. Like QuickHelp files, RTF files can contain formatting attributes and cross references.

Using the HelpMake Command

The HelpMake program can encode or decode Help files, allowing you to create new Help files or modify existing ones. Encoding converts a text file to a compressed Help database. HelpMake can encode text files written in QuickHelp, RTF, and minimally formatted ASCII format. Decoding converts a Help database to a text file for editing. HelpMake always decodes a Help database into a QuickHelp format text file.

The HelpMake command uses the following syntax:

```
HelpMake

   [Args]     [options] {/En | /Dn} {sourcefiles}
```

The *options* modify the action of HelpMake, and are described under HelpMake Options.

You must supply either option /E (encode) or option /D (decode). When encoding (/E) to create a Help database, you must use the /O option to specify the file name of the database.

The *sourcefiles* field is required. It specifies the input file for HelpMake. If you use the /D (decode) option, *sourcefiles* can be one or more Help database files (such as QH.Hlp). HelpMake decodes the database files into a single text file. If you use the /E (encode) option, *sourcefiles* can be one or more Help text files (such as QH.Src). Separate file names with a space. You can use standard wildcard characters.

The example below invokes HelpMake with the /V, /E, and /O options (see the heading Options for Encoding). HelpMake reads input from the text file my.txt and writes the compressed Help database in the file my.hlp. The /E option causes maximum compression. Note that the redirection symbol (>) sends a log of HelpMake activity to the file my.log. You may find it helpful to redirect the log file because, in its more verbose modes (given by /V), HelpMake may generate a lengthy log.

```
HelpMake

   [Args]     /V /E /Omy.hlp my.txt >my.log
```

The example below invokes HelpMake to decode the Help database my.hlp into the text file my.src, given with the /O option. Once again, the /V option results in verbose output, and the output is directed to the log file my.log. For more information, refer to the heading Options for Decoding.

```
HelpMake

   [Args]     /V /D /Omy.src my.hlp >my.log
```

HelpMake Options

HelpMake accepts a number of command-line options. These options fall into two categories:

- Encoding
- Decoding

You can specify options in uppercase or lowercase letters, and precede them with either a forward slash (/) or a dash (-). For example, -L, /L, -l, and /l all represent the same option. Most options apply only to encoding, others apply only to decoding, and a few apply to both.

Encoding Options

When you encode a file--build a Help database--you must specify the /E option. In addition, you can supply various other options that control the way HelpMake works. All the options that apply when encoding appear in Table 11-1.

Table 11-1. Options for Encoding

Option	Action
/A*c*	Specifies *c* as an application-specific control character for the Help database file. The character marks a line that contains special information for internal use by the application.
/C	Indicates that the context strings for this Help file are case sensitive. At run time, all searches for Help topics are case sensitive if the Help database was built with the /C option in effect.
/E[*n*]	Creates (encodes) a Help database from a specified text file. The optional *n* indicates the amount of compression to take place. If *n* is omitted, HelpMake compresses the file as much as possible, thereby reducing the size of the file by about 50%. The more compression requested, the longer HelpMake takes to create a database file. The value of *n* is a number in the range 0-15. It is the sum of successive powers of 2 representing various compression techniques, as listed here:

Value	Technique
0	No compression
1	Run-length compression
2	Key word compression
4	Extended key wordcompression
8	Huffman compression

continued

Table 11-1. Options for Encoding (cont.)

Option	Action
	Add values to combine compression techniques. For example, use /E3 to get run length and key word compression. This is useful in the testing stages of creating a Help database when you need to create the database quickly and are not concerned with size.
/H or /HELP	Displays a summary of HelpMake syntax and exits.
/K*filename*	Optimizes key word compression by supplying a list of characters that act as word separators. The *filename* is a file containing your list of separator characters. When you select key word compression, HelpMake scans the Help file to identify key words. A key word is any word that occurs often enough to justify replacing it with a shorter character sequence. HelpMake normally uses the following characters as word separators: •All characters from 0-32 (including the space) •!"#&'()*+'-,/:;<=>?@[\]^_'{|}~ •127 When performing key word compression, HelpMake treats as a word any series of characters not appearing in the separator list. Depending on the content of your Help file, you may be able to improve key word compression by using the /K option to specify a different list of separator characters. For instance, the default separator list contains the number sign (#). If your Help file contains #include directives, HelpMake normally treats #include as the word include without a number sign. To cause HelpMake to treat #include as a word, you could specify the following separator list: ! " & ' ( ) * + ' - , / : ; < = > ? @ [\] ^ _ ' { | } ~

continued

Table 11-1. Options for Encoding (cont.)

Option	Action
	The list does not include the number sign. HelpMake always treats characters in the range 0-32 as separators, so you do not need to include them. Your list must include all the other characters you want HelpMake to use as separators, including the space.
/L	Locks the generated file so that HelpMake cannot decode it at a later time.
/O*outfile*	Specifies *outfile* as the name of the Help database. You can specify a path, but your file name is limited to 14 characters.
/Sn	Specifies the type of input file, according to the following n values: **Option** **File Type** /S1 Rich Text Format (RTF) /S2 QuickHelp (default) /S3 Minimally formatted ASCII
/T	Translates dot commands into internal format. If your Help file contains dot commands other than .context, you should supply this option when encoding it. Dot commands are described in QuickHelp Format.

continued

Table 11-1. Options for Encoding (cont.)

Option	Action
/V[*n*]	Indicates the verbosity of diagnostic and informational output, depending on the value of *n*. Increasing the value adds more information to the output. If you specify only /V, HelpMake gives you its most verbose output. The possible values of *n* are listed here:

Option	Effect
/V	Maximum diagnostic output
/V0	No diagnostic output and no banner
/V1	Prints only HelpMake banner and fatal errors (default)
/V2	Prints pass names
/V3	Prints contexts on first pass
/V4	Prints contexts on each pass
/V5	Prints any intermediate steps within each pass
/V6	Prints statistics on Help file and compression

/W*width*	Indicates the fixed width of the resulting Help text in number of characters. The values of *width* can range from 11 to 255. If the /W option is omitted, the default is 76. When encoding RTF source (/S1), HelpMake automatically formats the text to *width*. When encoding QuickHelp (/S2) or minimally formatted ASCII (/S3) files, HelpMake truncates lines to this width.

Decoding Options

To decode a Help database into QuickHelp files, you must use the /D option. In addition, HelpMake accepts other options to control the decoding process. Table 11-2 shows all the options that are valid when decoding.

Table 11-2. Options for Decoding

Option	Action
/D[*letter*]	Decodes the input file into its original text or component parts. If a destination file is not specified with the /O option, the Help file is decoded to stdout. HelpMake decodes the file differently depending on the letter specified: ***Note:*** *For /D and /Du, if you decode a database created from multiple files (concatenated with APPEND), HelpMake decodes the first file only, and no message appears. When in doubt, use /DS first.* **Letter** **Effect** /D Decode Fully decodes the Help database, leaving all cross references and formatting information intact. /DS Decode split Splits the concatenated, compressed Help database into its components using their original names. If the database was created without concatenation (the default), HelpMake simply copies it to a file with its original name. No decompression occurs. Incompatible with /O and /T. /DU Decode unformatted Decompresses the database and removes all screen formatting and cross references. The output can still be used later for input and recompression, but all screen formatting and cross references are lost.

continued

Table 11-2. Options for Decoding (cont.)

Option	Action
/H or /HELP	Displays a summary of HelpMake syntax and exits without encoding or decoding any files.
/O*destfile*	Specifies *destfile* for the decoded output from HelpMake. If *destfile* is omitted, the Help database is decoded to stdout. HelpMake always decodes Help database files into QuickHelp format.
/T	Translates dot commands from internal format into dot-command format. You should always supply this option when decoding a Help database that contains dot commands other than .context.
/V[*n*]	Indicates the verbosity of diagnostic and informational output depending on the value of *n*. If you specify only /V, HelpMake gives you its most verbose output. The possible values of *n* are listed here:

Option	Effect
/V	Maximum diagnostic output
/V0	No diagnostic output and no banner
/V1	Prints only the HelpMake banner and fatal errors (default)
/V2	Prints pass names
/V3	Prints contexts on first pass

Modifying a Help Database

You can modify a CTOS Microsoft-compatible Help database using two methods. The first method involves decoding, modifying, and encoding a Help database. The second method involves splitting a concatenated database into component Help files, and then decoding, modifying, encoding, and reassembling the database.

The first method is the simplest, and follows these steps, using the file mLINK.Hlp as an example:

1. Decode MLINK.Hlp with this command:

   ```
   HelpMake

       [Args]  /D /T /OMLINK.Src  MLINK.Hlp
   ```

2. Edit MLINK.src with the CTOS Editor. Be sure to Set Defaults (F7) for RLF (Carriage Return Line Feeds). This is a required setting. (All lines must end with the triangle and down arrow.)

3. Encode the changed source file using this command:

   ```
   HelpMake

        [Args]  /E /T /W77 /OMLINK.Hlp  MLINK.Src
   ```

4. Test your results.

   Note: *MLINK.Hlp is a file found in <MSCHelp>. If you practice with it, be sure to practice with a copy.*

The second method is more complex, edits one file of many that create one database, and follows these steps using the file UTILS.Hlp as an example:

1. Split UTILS.Hlp with this command:

```
HelpMake

    [Args]  /Ds UTILS.Hlp
```

   This divides UTILS.Hlp into 4 Help files that appear in your working directory: MISC.Hlp, NMAKE.Hlp, HELPMAKE.Hlp, and LIB.Hlp.

2. Decode NMAKE.Hlp with this command:

```
HelpMake

    [Args]  /D /T /ONMAKE.Src  NMAKE.Hlp
```

3. Edit NMAKE.src with the CTOS Editor. Be sure to Set Defaults (F7) for CRLF (Carriage Return Line Feeds). This is a required setting. (All lines must end with the triangle and down arrow.)

4. Encode the changed source file using this command:

```
HelpMake

    [Args]  /E /T /W77 /ONMAKE.Hlp NMAKE.Src
```

5. Delete the existing UTILS.Hlp, then recreate UTILS.Hlp with this command:

```
APPEND

    File list from      MISC.Hlp NMAKE.Hlp HELPMAKE.Hlp
                        LIB.Hlp
    File to             UTILS.Hlp
```

6. Test your results.

   Note: *UTILS.Hlp is a file found in <MSCHelp>. If you practice with it, be sure to practice with a copy in a separate directory.*

The above examples illustrate the commands and switches used to create the Help files that accompany CTOS Microsoft C. For an example of a makefile that creates Utils.Hlp, refer to the first example in Appendix B.

Help Text Conventions

CTOS Microsoft Help databases have a common structure and follow certain organizational conventions. You should follow the same conventions to create Microsoft-compatible Help files. These conventions fall into four categories:

- File Structure
- Local Contexts
- Context Prefixes
- Hyperlinks

Help Text File Structure

The Help retrieval capability that is built into Microsoft products is simply a data retrieval tool. It imposes no restrictions on the content and format of the Help text. The HelpMake utility and the display routines built into Microsoft language environments, however, make certain assumptions about the format of Help text. This subsection provides some guidelines for creating Help text files compatible with those assumptions.

In all three Help text formats, the Help text source file is a sequence of topics, each preceded by one or more unique context definitions. The following list specifies the various formats and the corresponding context definition statements:

Format	Context Definition
QuickHelp	.context *context*
Minimally formatted	>>*context*
ASCII	(none)
RTF	\ par>>*context*\ par

In QuickHelp format, each topic begins with one or more .context statements that define the context strings that map to the topic text. Subsequent lines up to the next .context statement constitute the topic text.

In minimally formatted ASCII, each context definition must be on a separate line, and each must begin with the Help delimiter (>>). As in RTF and QuickHelp files, subsequent lines up to the next context definition constitute the topic text.

In RTF format, each context definition must be in a paragraph of its own (denoted by \ par), beginning with the Help delimiter (>>). Subsequent paragraphs up to the next context definition constitute the topic text.

For detailed information about these three formats, refer to the heading Using Help Database Formats.

Local Contexts

Context strings that begin with an at sign (@) are defined as local and have no implicit cross references. They are used in cross references instead of the context string that otherwise is generated.

When you use a local context, HelpMake does not generate a global context string (a context string that is known throughout the Help file). Instead, it embeds an encoded cross reference that has meaning only within the current context. For example:

```
.context normal
```

This is a normal topic, accessible by the context string normal.

[button\ v@local\ v] is a cross reference to the following topic.

```
.context @local
```

You can reach this topic only if you browse sequentially through the file or use the cross reference in the previous topic.

In this example, the text [button\ v@local\ v] defines local as a local context. If the user selects the text [button] or scrolls through the file, the Help system displays the topic text that follows the context definition for local. Because local is defined with the at sign (@), it can be accessed only by a hyperlink within the Help file or by sequentially browsing through the file. Making a context local saves file space and speeds access.

Context Prefixes

Microsoft Help databases use several context prefixes. A context prefix is a single letter followed by a period. It appears before a context string that has a predefined meaning. If you decode a Microsoft Help database, many of these contexts may appear in the resulting text file.

Except for the h. prefix, described here, context prefixes are internal. You do not need to add them in Help files that you write.

You can use the h. prefix to identify standard help-file contexts. For instance, h.default identifies the default Help screen: the screen that normally appears when you select top-level help. Table 11-3 lists the standard h. contexts.

Table 11-3. Standard h. Contexts

Context	Description
h.contents	The table of contents for the Help file. You should also define the string contents for direct reference to this context.
h.default	The default Help screen, typically displayed when the user presses F1 at the top level in most applications. The contents are generally devoted to information about using Help.
h.index	The index for the Help file. You can also define the string index for direct reference to this context.
h.notfound	The Help text that displays when the Help system cannot find information about the requested context. The text could be an index of contexts, a topical list, or general information about using Help.

continued

Table 11-3. Standard h. Contexts (cont.)

Context	Description
h.pg#	A specific page within the Help file. This is used in response to a go to page # request.
h.pg$\	The Help text that is logically last in the file. Some applications use this in response to a go to the end request made within the Help window.
h.pg1	The Help text that is logically first in the file. Some applications use this in response to a go to the beginning request made within the Help window.
h.title	The title of the Help database.

The context prefixes in Table 11-4 are internal to Microsoft products. They appear in decompressed databases, but you do not need to use them.

Table 11-4. Microsoft Product Context Prefixes

Prefix	Purpose
d.	**Dialog box** Each dialog box is assigned a number. Its Help context string is d. followed by the number (for example, d.12).
e.	**Error number** If a product supports the error-numbering scheme Microsoft languages use, it displays Help for each error using this prefix. For example, the context e.c1234 refers to the C compiler error message number C1234.
m.	**Menu item** Accelerator keys define contexts that relate to product menu items. For example, SHIFT-F X acesses the Exit selection on the File menu item, so m.f.x references it in Help.
n.	**Message number** Each message box is assigned a number. Its Help context string is n. plus the number (for example, n.5).

Hyperlinks

Explicit cross references, or hyperlinks, in the Help text file are marked with invisible text. A hyperlink comprises a word or phrase followed by invisible text that gives the context to which the hyperlink refers.

The keystroke that activates the hyperlink depends on the application. Consult the documentation for each product to find the specific keystroke needed.

When the user activates the hyperlink, the Help system displays the topic named by the invisible text. The invisible cross reference text is formatted as one of the following:

Hyperlink Text	Action
contextstring	Causes the Help topic associated with *contextstring* to display. For example, exeformat results in the display of the Help topic associated with the context exeformat.
filename!	Treats *filename* as a single topic to display. For example, $INCLUDE:stdio.h! searches the INCLUDE environment variable for file STDIO.h and displays it as a single Help topic.

filename!contextstring

Works the same way as *contextstring*, except that only the Help file *filename* is searched for the context. If the file is not already open, the Help system finds it (by searching either the current path or an explicit environment variable) and opens it. For example, $BIN:readme.doc!patches searches for readme.doc in the BIN environment variable and displays the topic associated with patches.

In the following example, the word Example is a hyperlink:

```
\bSee also:\p   \uExample\p\vopen.ex\v
```

The hyperlink refers to open.ex. If you select any of the letters of Example, the Help system displays the topic whose context is open.ex. On the screen, this line appears as follows:

```
See also:   Example
```

An application might display See also: and Example in different colors or character types, depending on such factors as your default color selection and type of monitor.

When a hyperlink needs to cross reference more than one word, you must use an anchor, as in the following example:

```
\bSee also:\p   \uExample\p\vprintf.ex\v, fprintf, scanf,
sprintf, vfprintf, vprintf, vsprintf
             \aformatting table\vprintf.table\v
```

This part of the example is an anchored hyperlink:

```
\aformatting table\vprintf.table\v
```

The \a flag creates an anchor for the cross reference. In the example, the phrase following the \a flag (formatting table) is the hyperlink. It refers to the context printf.table. The first \v flag marks both the end of the hyperlink and the beginning of the invisible text. The name printf.table is invisible; it does not appear on the screen when the Help displays. The second \v flag ends the invisible text.

Using Help Database Formats

Database text format can be any of three types:

- QuickHelp
- ASCII
- Rich Text

The following list briefly describes these types.

Type	Characteristics
QuickHelp	Uses dot commands and embedded formatting characters (the default formatting type expected by HelpMake); supports highlighting, color, and cross references. This format requires compression before use.
Minimally formatted ASCII	Uses a Help delimiter (>>) to define Help contexts; does not support highlighting, color, or cross references. This format does not require compression.
RTF	Uses a subset of standard RTF; supports highlighting, color, and cross references; supports dot commands. This format requires compression.

An entire Help system can use any combination of files formatted with different format types. With CTOS Microsoft C, for example, the README.DOC information file is encoded as unformatted ASCII, and the Help files for the C language and run-time library are encoded in the QuickHelp format. The database also cross references the header (include) files, which are unformatted ASCII files stored outside the database.

QuickHelp Format

To convey information to HelpMake, the QuickHelp format uses three components:

- Dot Commands
- Formatting Flags
- cross references

QuickHelp Dot Commands

QuickHelp supports a number of dot commands, which identify topics and convey other topic-related information to the Help system. If your Help file contains dot commands other than .context, you must supply the /T option when encoding and decoding with HelpMake.

The most important dot command is the .context command. Every topic in a QuickHelp file begins with one or more .context commands. Each .context command defines a context string for the topic text. You can define more than one context for a single topic, as long as you do not place any topic text between them.

Typical dot commands are shown below. The first defines a context for the #include C preprocessor directive. The second set illustrates multiple contexts for one block of topic text. In this case, the same topic text explains all of the string-to-number conversion routines in C.

```
.context #include
       .
       .  description of #include goes here
       .
 .context strtod
 .context strtol
 .context strtoul
       .
       .  description of string-to-number functions goes here
```

The QuickHelp format supports several other dot commands. Table 11-5 lists all of the dot commands available in QuickHelp format. In addition, Table 11-5 also lists equivalent colon commands, which you may find if you browse .Hlp files.

Table 11-5. QuickHelp Dot Commands

Command	Action
.category *string* :c*string*	Lists the category in which the current topic appears and its position in the list of topics. The QuickHelp Topic command uses the category name to dsiplay the list of topics containing the current topic. Some applications, such as PWB, use this name as a pointer to the applicable table of contents.
.command :x	Indicates that the topic text is not a displayable Help topic. Use this command to hide hyperlink topics and other internal information. Hyperlink topics are described in Hyperlink Commands.
.comment *string*	The *string* is a comment that appears only in the Help source file. Comments are especially useful for documenting the purpose of cross references. HelpMake ignores comments in a Help source file and does not insert them into the Help database.
.context *string*	The *string* introduces a topic.
.end :e	Ends a paste section. See the .paste command.
.freeze *numlines* :z*numlines*	Indicates that the first *numlines* lines should be frozen as the top line of the Help screen. This is normally used to freeze a row of cross reference buttons at the top of a Help topic that might be scrolled.
.length *topiclength* :l*topiclength*	Indicates the default window size, in *topiclength* lines, of the topic about to display. This command is always the first line in the topic if present.

continued

Table 11-5. QuickHelp Dot Commands (cont.)

Command	Action
.list :i	Indicates that the current topic contains a list of topics. QuickHelp displays a highlighted line; you can choose a topic by moving the highlighted line over the desired topic and pressing RETURN. Help searches for the first word of the line.
.mark *name* [*column*] :m*name* [*column*]	Defines a mark immediately preceding the following line of text. You can use this command in Help script commands to indicate that the display of a particular topic begins at the marked line. The *name* identifies the mark. The optional *column* value is an integer that indicates a column location within the specified line.
.next *context* :>*context*	Tells the Help system to look up the next topic using *context* instead of the next topic's name. You can use this command to skip large blocks of .command or .popup topics.
.previous *context* :<*context*	Tells the Help system to look up the previous topic using *context* instead of the previous topic's name. You can use this command to skip large blocks of .command or .popup topics.
.paste *pastename* :p*pastename*	Begins a paste section. The *pastename* appears in the QuickHelp Paste menu.
.popup :g	Tells the Help system to display the current topic as a popup instead of a normal, scrollable topic.
.ref *string(s)* :r*string(s)*	Tells the Help system to display the list of *string* topics in the Reference menu. You can list as many topics as needed; separate each additional *string* with a comma.
.topic *text* :t*text*	Defines *text* as the name or title to display in place of the context string if the application Help displays a title. This command is always the first line in the context unless you also use the .length command.

QuickHelp Formatting Flags

The QuickHelp format supports a number of formatting flags that are used to highlight parts of the Help database and to mark hyperlinks in the Help text.

Each formatting flag consists of a backslash (\ ) followed by a character. Table 11-6 lists the formatting flags.

Table 11-6. Formatting Flags

Formatting Flag	Action
\a	Anchors text for cross references
\b, \B	Turns boldface on or off
\p, \P	Turns off all attributes
\u, \U	Turns underlining on or off
\v, \V	Turns invisibility on or off (hides cross references in text)
\\	Inserts a single backslash in text

On monochrome monitors, text labeled with the italic attribute appears in various ways (for example, high intensity and reverse video are commonly displayed). On color monitors, the application translates these attributes into suitable colors.

The \b, \i, \u, and \v options are toggles, turning on and off their respective attributes. You can use several of these on the same text. Use the \p attribute to turn off all attributes. Use the \v attribute to hide cross references and hyperlinks in the text.

HelpMake truncates the lines in QuickHelp files to the width specified with the /W option. (For more information, refer to the heading Options for Encoding.) Only visible characters count toward the character-width limit. Lines that begin with an application-specific control character are truncated to 255 characters regardless of the width specification. For details about application-specific control characters refer to the heading Options for Encoding.

In the following example, the \b flag initiates boldface text for Returns:, and the \p flag that follows the word reverts to plain text for the remainder of the line.

```
\bReturns:\p a handle if successful, or -1 if not.
        errno: EACCES, EEXIST, EMFILE, ENOENT
```

In the following example, \a anchors text for the hyperlink Example. The \v flags define the cross reference to be sample_prog and cause the text between the flags to be invisible.

```
\aExample \vsample_prog\v
```

Cross references are described in the following subsection.

QuickHelp Cross References

Help databases contain two types of cross references: implicit and explicit. They are described in Contents of a Help File.

An implicit cross reference is any word that appears both in the topic text and as a context in the Help file. For example, any time you request Help on the word close, the Help window displays Help on the close function. You do not need to code implicit cross references in your Help text files.

Explicit cross references (hyperlinks) are words or phrases on the screen that are associated with a context. For example, the word Example in the initial Help-screen area for any C function is an explicit cross reference to the C program example for that function. You must insert formatting flags in your Help text files to mark explicit cross references.

If the hyperlink consists of a single word, you can use invisible text to flag it in the source file. The \v formatting flag creates invisible text, as follows:

```
hyperlink\vcontext\v
```

Specify the first \v flag immediately following the word you want to use as the hyperlink. Following the flag, insert the context that the hyperlink cross references. The second \v flag marks the end of the context, that is, the end of the invisible text. HelpMake generates a cross reference whose context is the invisible text, and whose hyperlink is the entire word.

If the hyperlink consists of a phrase, rather than a single word, you must use anchored text to create explicit cross references. Use the \a and \v flags to create anchored text as follows:

```
\ahyperlink-words\vcontext\v
```

The \a flag marks an anchor for the cross reference. The text that follows the \a flag is the hyperlink. The hyperlink must fit entirely on one line. The first \v flag marks both the end of the hyperlink and the beginning of the invisible text that contains the cross reference context. The second \v flag marks the end of the invisible text.

The following example contains three implicit cross references to the C routines abs, cabs, and fabs:

```
See also: abs, cabs, fabs
```

The following example shows the encoding for an explicit cross reference to an example program and a function template from the Help database for the C run-time library:

```
See also: Example\vopen.ex\v, Template\vopen.tm\v, close
```

Here, the hyperlinks are Example and Template, which reference the contexts open.ex and open.tm. The example also contains an implicit cross reference to the close function.

The following example shows the encoding for an explicit cross reference to an entire family of functions:

```
See also: \ais...  functions\vis_functions\v, atoi
```

The cross reference uses anchored text to associate a phrase, rather than just a word, with a context. In this example, the hyperlink is the anchored phrase "is... functions", and it cross references the context is_functions. In addition, the example contains an implicit cross reference to the atoi routine.

The code below is an example in QuickHelp format that contains a single entry:

```
.context open
 .length 13
 \bInclude:\p <fcntl.h>, <io.h>, <sys\\types.h>,<sys\\stat.h>

 \bPrototype:\p int open(char *path, int flag[, int mode]);
       flag: O_APPEND    O_BINARY  O_CREAT O_EXCL  O_RDONLY
             O_RDWR      O_TEXT    O_TRUNC  O_WRONLY
               (can be joined by \v)
       mode: S_IWRITE    S_IREAD   S_IREAD \v S_IWRITE

 \bReturns:\p  a handle if successful, or -1 if not.
          errno: EACCES, EEXIST, EMFILE, ENOENT

 \bSee also:\p  \uExample\p\vopen.ex\v,
\uTemplate\p\vopen.tp\v,
       access, chmod, close, creat, dup, dup2, fopen, sopen,
umask
```

The .length command near the beginning of the example specifies the size of the initial window for the Help text. Here, the initial window displays 13 lines.

The manifest constants (such as O_WRONLY and EEXIST), the C keywords (such as int and char), and the other functions (such as sopen and access) are implicit cross references. The words Example and Template are explicit cross references to the example open.ex and to the open template open.tp, respectively. Note the use of double backslashes in the include file names.

Minimally Formatted ASCII Format

A minimally formatted ASCII text file comprises a sequence of topics, each preceded by one or more unique context definitions. Each context definition must be on a separate line beginning with a Help delimiter (>>). Subsequent lines up to the next context definition constitute the topic text.

You can use minimally formatted ASCII files in two ways. You can compress the file with HelpMake, creating a Help database, or an application can access the uncompressed file directly. Uncompressed files are somewhat larger and slower to search, however. Minimally formatted ASCII files are of fixed width, and they cannot contain highlighting (or other nondefault attributes) or cross references.

The following example, coded in minimally formatted ASCII, shows the same text as the QuickHelp example in the previous subsection. The first line of the example defines open as a context string. The minimally formatted ASCII Help file must begin with the Help delimiter (>>), so that HelpMake or the application can verify that the file is indeed an ASCII Help file.

```
>>open

 Include: <fcntl.h>, <io.h>, <sys\types.h>, <sys\stat.h>

 Prototype: int open(char *path, int flag[, int mode]);
       flag: O_APPEND O_BINARY O_CREAT O_EXCLO_RDONLY
             O_RDWR   O_TEXT   O_TRUNC O_WRONLY
          (can be joined by \v)
       mode: S_IWRITE S_IREAD  S_IREAD \v S_IWRITE

 Returns:  a handle if successful, or -1 if not.
      errno: EACCES, EEXIST, EMFILE, ENOENT

 See also:  access, chmod, close, creat, dup, dup2, fopen,
sopen, umask
```

When displayed, the Help information appears exactly as it is typed into the file. Any formatting codes are treated as ASCII text. Note that you do not need to escape backslashes in minimally formatted ASCII files.

If you compress minimally formatted ASCII files, they are smaller and faster to search.

Rich Text Format (RTF)

RTF is a Microsoft word processing format supported by many other word processors. It allows documents to be transferred from one application to another without losing any formatting information. The HelpMake utility recognizes a subset of the full RTF syntax. If your file contains any RTF code that is not part of the subset, HelpMake ignores the code and strips it out of the file.

Certain word processing and file-conversion programs generate the RTF code automatically as output. You need not worry about inserting RTF codes yourself; you can simply format your Help files directly with a word processor that generates RTF, using the attributes supported by the subset. The only items you need to insert are the Help delimiter (>>) and context string that start each entry.

HelpMake recognizes the subset of RTF listed below:

RTF Code	Action
\ b	Boldface
	The application decides how to display this; often it is intensified text.
\ fi <*nnn*>	Paragraph first-line indent
\ i	Italic
	The application decides how to display this; often it is reverse video.
\ li <*nnn*>	Paragraph indent from left margin
\ line	New line (not new paragraph)
\ par	End of paragraph

RTF Code	Action
\ pard	Default paragraph formatting
\ plain	Default attributes On most screens this is nonblinking normal intensity.
\ tab	Tab character
\ ul	Underline The application decides how to display this; some adapters that do not support underlining display it as blue text.
\ v	Hidden text Hidden text is used for cross reference information and for some application-specific communications; it is not displayed.

Using the word processing program, you can break the topic text into paragraphs. When HelpMake compresses the file, it formats the text to the width given with the /W option, ignoring the paragraph formats.

As with the other text formats, each entry in the database source consists of one or more context strings, followed by topic text. An RTF file can contain QuickHelp dot commands.

The Help delimiter (>>) at the beginning of any paragraph denotes the beginning of a new Help entry. The text that follows on the same line is defined as a context for the topic. If the next paragraph also begins with the Help delimiter, it also defines a context string for the same topic text. You can define any number of contexts for a block of topic text. The topic text comprises all subsequent paragraphs up to the next paragraph that begins with the Help delimiter.

The code below is an example of a Help database that contains a single entry using subset RTF text. Note that RTF uses curly braces ({ }) for nesting. Thus, the entire file is enclosed in curly braces, as is each specially formatted text item.

```
{\rtf1
 \pard >>open\par
   {\b Include:}   <fcntl.h>, <io.h>, <sys\\types.h>,\par
   <sys\\stat.h>\\ par
 \par
   {\b Syntax:}   int open(char * filename, int oflag[, int\par
   pmode]);\par
        oflag: O_APPEND O_BINARY O_CREAT O_EXCL_par O_RDONLY\p
               O_RDWR O_TEXT O_TRUNC O_WRONLY\par
    (may be joined by \v)\par
        pmode: S_IWRITE S_IREAD S_IREAD \vS_IWRITE\par
 \par
   {\b Returns:}   a handle if successful, or -1 if not.\par
   errno: EACCES, EEXIST, EMFILE, ENOENT\par
 \par
   {\b See also:}   Examples{\v open.ex}, access, chmod, \par
   close, creat, dup, dup2, fopen, sopen, umask\par
 >>>open.ex\par
 To build this Help file, use the following command:\par
 \par
 HelpMake /S1 /E15 /OOPEN.HLP OPEN.RTF\par
 \par
        < Back >{\v !B}
 }
```

Actual RTF output normally contains additional information that is not visible to the user; HelpMake ignores this extra information.

Advisor Help Library

The Advisor Help library contains a procedural interface that you can use to add or modify Help text to an application's Help system.

You can build a Help system for your application by using:

- the procedural interface to call functions that your Help databases use
- HelpMake to create the Help databases

The interface calls are described in the Microsoft C online Help.

If you are using the Advisor, refer to Advisor Help Library in the HelpMake Table of Contents.

If you are using QuickHelp, search (press S) for the topic, Advisor Help Library.

Glossary

This glossary defines terms used in this book, and CTOS-specific terms that programmer's commonly encounter.

A

ANSI Standard.
American National Standards Institute Standard for the C Language.

application partition.
A section of user memory reserved to execute applications. A workstation can have multiple partitions with concurrently running applications.

archive file.
A file created by one of the backup commands (for example, Backup Volume, Selective Backup, or Tape Backup). Archive files store the contents of other files in a compressed format.

archive media.
Floppy diskettes, QIC tapes, or half-inch tapes used to store archive files.

arithmetic operators.
Symbols used in arithmetic expressions that indicate the operation to perform. The standard operators are add (+), subtract (-), multiply (*), and divide (/).

array declarator.
A declarator followed by square brackets, possibly enclosing an integral constant expression. If no expression appears, the array has unknown size; otherwise, the expression is the number of elements in the array.

ASCII.
An acronym for the American Standard Code for Information Interchange. ASCII defines the character set codes used for information exchange between equipment. ASCII commonly denotes a file containing unformatted text.

_asm block.
A code block that starts with the _asm keyword, followed by an assembly instruction or group of instructions, possibly enclosed in braces.

Assembly Language.
The lowest-level language you can use to write CTOS programs. You can use CL to convert assembly files into CTOS object modules.

B

Bind.
The Executive command that activates the Linker to create a Version 6 run file. Also the act of creating a Version 6 run file. Version 6 run files are required for protected mode compatibility.

BNet II.
Communications software program allows users to access the system files and other resources in remote locations. It includes utilities for data transfer and network administration.

boot.
The operation that starts a system when powered on or reset. The boot operation runs self-diagnostic tests, loads the operating system, and reads system files.

braces ({ }).
The characters used to enclose a network node name in a file specification or path setting.

BTOS II.
A high performance Unisys operating system for B24, B26, B27, B28, B38, and B39 workstations and shared resource processors. See also CTOS.

buffer.
An area of memory used for temporary storage of input and output data.

byte.
A unit of computer information, usually composed of eight bits, and often representing one character.

C

C.
A high-level language used to write CTOS programs. You can use the C compiler to convert the programs into object modules.

calling convention.
The way a language implements a call. The choice of calling convention affects the machine instructions that a compiler generates to execute and return from a function, procedure, or subroutine call; and the order in which the function arguments are passed (right to left or left to right).

case sensitive.
A parameter, language, or command that requires text to match a certain pattern exactly, including all uppercase and lowercase letters. Most CTOS commands are not case sensitive; that is, the operating system interprets COPY and copy the same way.

cast.
An abstract type enclosed in parentheses. Casts can appear after a sizeof keyword, or be used as a unary operator to convert the operand expression to the named type. Both the operand and the cast must have scalar type.

CL.
The compiler command supplied with CTOS Microsoft C.

client.
A general term for a user, software application, or computer that requires the services, data, or processing of another application or computer.

client process.
A process that requests a system service. All processes, even built-in operating system processes, can be client processes, since all processes can request system services.

cluster.
One or more workstations connected to a shared resource processor or server workstation.

cluster configuration.
A local resource-sharing network consisting of a server connected to cluster workstations. Each workstation has one high-speed RS-422 channel that enables intercluster communication. Large clusters with a shared resource processor may require multiple RS-422 channels. See also cluster workstation, server, and server workstation.

cluster workstation.
A workstation connected to a server workstation in a cluster configuration. A cluster workstation optionally has local file storage. See also cluster configuration and server.

COBOL (COmmon Business Oriented Language).
One of the high-level languages you can use to write CTOS programs. You can use the COBOL Compiler to convert source code into object modules.

code listing.
A readable display of compiled code.

code segment.
A variable-length (up to 64K) logical entity consisting of re-entrant code, and containing one or more complete procedures.

COMDEF OMF record.
COMmunal DEFinition, Object Module Format record that CTOS Microsoft C emits to the object file for uninitialized global variables.

command file.
A file the Executive reads to display command names, Executive command forms, and Help descriptions. The Executive associates a run file and command case with each command.

compiler.
A program that translates high-level language programs into object modules (machine-readable code).

configuration file.
A file that contains parameter values for a software product or hardware device. A configuration file specifies data used by utilities, applications, and the operating system.

Context Manager.
A software program that divides memory into multiple partitions in which you can start and concurrently run applications and utilities.

corrupted volume.
An initialized disk no longer recognized by the operating system.

CTOS.
A high performance Unisys operating system for B24, B26, B27, B28, B38, and B39 workstations and shared resource processors. CTOS is also an umbrella term encompassing all varieties of the BTOS, CTOS, and CTOS/XE operating systems.

CTOS.Lib.
Part of the Language Development software. It is a library of object modules that provide operating system run-time support.

D

default data segment.
See DGroup.

default value.
A value used by commands when optional fields are blank.

denormalized numbers.
Nonzero floating-point numbers with reserved exponent values in which the most-significant bit of the mantissa is zero.

dependency tree.
NMAKE's system of building files in the order needed to update the original target. NMAKE never builds targets until it updates (recompiles) all changed files.

dependent.
A file used to create a target, such as a C source file. Dependents are listed in makefiles.

description block.
Code that tells NMAKE how to build a project's target files. A description block specifies targets, dependents, and commands.

description file.
A description block plus comments, macros, inference rules, and directives, found in a makefile.

DGroup.
DGroup usually includes data, constant, and stack segments. It is also referred to as the default data segment.

directory specification.
The identifier for a directory. It consists of a node name, volume name, and a directory name.

distributed processing.
Using multiple processors to achieve a single result.

dynamically-installed system service
A program that loads as an application program, and then uses the CTOS ConvertToSys operation to convert itself into a system service. Once installed, a dynamically-installed system service has the same capabilities as a system service that was linked with the System Image during system build. A dynamically-installed system service uses operating system operations to identify the request codes that it services, specify its execution priority, establish its interrupt handlers, and so forth.

E

Envelope function.
For CTOS Microsoft 6.0, contains code to save and restore the SI and DI registers and clear the direction flag. You must link envelope function objects with your program in addition to the currently linked libraries. With CTOS Microsoft C 6.1, the _plm keyword generally makes these functions unnecessary.

erc.
See status code.

error code.
See status code.

error message.
A message that appears when an error occurs in an operating system, system service, or application. Problems that occur during bootstrap may appear as keyboard lights and audible beeps.

event.
Any external stimulus that causes an action to occur. The stimulus can be generated by users (keystrokes or mouse clicks), applications (unexpected messages), or operating systems (interval lapses in timer management).

Executive.
The CTOS command interpreter. The Executive is an interactive application program that requests the operating system to load programs that execute commands entered by the user. The Executive is usually loaded from the [Sys]<Sys>Exec.run file.

expansion.
The physical replacement of wild card characters for the actual characters they match.

exponent.
Second part of a floating-point number that contains the order of magnitude of the number. See also Mantissa.

expressions.
In a program, a combination of various constants, variables, operators, and parentheses used to perform a computation.

external reference.
A reference from one object module to variables and entry points of other object modules.

F

fh.
An acronym for file handle.

field.
An area that accepts parameter values in a command form.

file access methods.
Object module procedures located in the standard CTOS library. They provide buffering, and use the asynchronous input/output capabilities of the file management system to overlap input/output and computation.

file handle.
A 16-bit integer that uniquely identifies an open file. It is returned by the CTOS OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

file name.
A unique name that specifies a file. It can contain a maximum of 50 alphanumeric characters, including uppercase and lowercase letters, periods, hyphens, and right angle brackets. A file name is the fourth element of a full file specification.

file specification.
A unique identifier that contains the name of a file, as well as its volume, directory, and node location. A complete file specification has the form {Node}[Vol]<Directory>FileName^password.

file system.
The data and control structures stored on an accessible disk device.

flag.
An indicator that denotes the existence of a particular condition. It can have a value of TRUE or FALSE.

full file specification.
The node name, volume name, directory name, file name, and optional password for a file.

function declarator.
A declarator followed by an optional set of language modifiers and a pair of parentheses, possibly empty.

G

general protection fault.
A system fault generated by breaking any of the Protected mode protection rules. General protection faults occur when a process attempts to access memory that is not allocated to it.

group.
A named collection of Linker segments that the CTOS loader addresses at run time with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64K of each other.

I

identifier.
A sequence of uppercase and lowercase letters, digits, and the underscore character. An identifier must begin with a letter or the underscore. It can be any length, but the first 32 characters are significant only.

IEEE.
Institute of Electrical and Electronics Engineers.

inference rules.
Templates that NMAKE uses to create files with a given extension. Inference rules provide a convenient shorthand for common operations.

inherited macros.
Macros NMAKE creates that are equivalent to every current environment variable. These macros inherit the names and values of the corresponding environment variables.

initialization.
A process performed at the beginning of a program to ensure that all indicators and constants are set to prescribed conditions and values before you use the program.

interactive.
A program that users can manipulate.

J

Job Control Language (JCL).
A programming language processed by the Batch facility.

K

kernel.
The most primitive and the most powerful component of the operating system. The Kernel provides interprocess communication (IPC) primitives, and schedules process execution. It executes with a higher priority than any process, but it is not itself a process. See also process.

kilobyte.
1024 bytes of data or storage space.

L

Language Development.
The CTOS Language Development software that provides the Linker, CTOS Librarian, and CT Assembler programs (Bind, Librarian, and Assemble Executive commands).

.lib.
The standard file name suffix for library files.

library.
A collection of object modules (complete routines or subroutines) that can be linked into run files.

library file.
A file that contains one or more object modules. The file name normally includes the suffix .lib.

Link.
The Executive command that displays the Linker command form for combining a list of object modules into a version 4 run file. Also generically refers to the act of creating a run file.

linker.
A program that combines object modules into run files.

linker segment.
A single entity consisting of all segment elements with the same segment name.

loadable request file.
A binary file that contains request definitions for system services. The loadable request file merges new requests with the requests in Request.sys during installation of system services. When bootstrapped, the operating system reads Request.sys, loads it into memory, and adds the requests to the basic request routing table.

local context.
A Help file in which HelpMake embeds an encoded cross-reference that has meaning only within the current context (topic).

local file system.
The system software that manages disk access requests on cluster workstations with hard disks. The local file system allows cluster workstations to access files on local hard disks, as well as files on server hard disks. The filter process of the local file system intercepts each file access request and directs it to the local file system or to the server.

logical file address (lfa).
A 32-bit unsigned integer used to locate a particular sector of a file. An lfa specifies the byte position within a file, and is the number (offset) that would be assigned to a byte in a file if all the bytes were numbered consecutively starting with 0. For example, the lfa of the third sector of a file is 1024. An lfa must be on a sector boundary and is thus a multiple of 512.

M

.Map.
.Map is the standard file name suffix for Linker list files.

mantissa.
First part of a floating-point number that contains the value of the number. See also Exponent.

map file.
The Linker list file (suffix .Map) that contains an entry for each Linker segment, identifying the segment relative address and length in the memory image. You can direct the Linker to list public symbols and line numbers.

MCBIND
One of two command forms supplied with CTOS Microsoft C that you can use to invoke the CTOS Linker. See also MLINK.

megabyte (MB).
1,048,576 bytes of data or storage space.

message file.
A binary file containing the screen prompts and messages displayed by an application.

MLINK
One of two command forms supplied with CTOS Microsoft C that you can use to invoke the CTOS Linker. MLINK executes automatically after successful compiles. See also MCBIND.

N

naming convention.
The way a compiler alters a routine name before placing it in the object file. You can choose between C and Pascal naming conventions to ensure that the names in the calling program agree with those in the called program.

NAN.
Acronym for Not A Number.

NCP.
See numeric coprocessor.

network node.

A server connected to BNet II or CT-Net. Cluster workstations connected to a node can communicate with other network nodes. Some releases of BNet II also allow cluster workstations to be network nodes.

NMAKE.
The program maintenance utility that simplifies project management by determining which project files depend on others. NMAKE can automatically update your project when any project file has changed.

numeric coprocessor.
An Intel 8087, 80287, or 80387 chip.

O

.obj.
The standard file name suffix for object module files.

object module.
A file that results from a single compiler or assembler function. You can link the object module with other object modules into executable run files.

object module procedure.
A procedure supplied as part of an object module file. It is linked with the user-written object modules of an application program and is not supplied as part of the System Image. Most application programs require only a subset of these procedures. When the application program is linked, the desired procedures are linked together in the run file of the application.

optimization.
What the compiler does when it rewrites parts of a program to make it optimally efficient in ways that are not apparent at the source level.

output.
Data delivered from a program to a file or device.

P

parameter.
A variable or constant transferred to and from a subroutine or program, affecting program execution.

parameter value.
An element of information supplied in a command form, procedural call, or configuration file.

partition.
A defined portion of workstation memory. There are two types of partitions: system and application. Each application runs in a separate partition. A partition management program such as Context Manager allows multiple application programs to execute simultaneously, since each program restricts instructions and data to its assigned memory area.

partition management.
A program that provides operations for creating, managing, and removing application partitions. Partition management permits concurrent execution of multiple application programs, each in its own partition. See also partition managing program.

partition managing program.
A program, such as Context Manager, which uses partition management operations to manage multiple application partitions in memory. Each partition can contain an executing application program. See also partition management.

Pascal.
One of the high-level languages you can use to write CTOS programs. The Pascal Compiler converts pascal source programs into CTOS object modules.

physical address.
An address relative to memory location 0. It does not specify a segment base.

pointer.
An address that specifies a storage location for data.

pointer declarator.
A declaration beginning with an asterisk (*).

processor.
The central processing unit (CPU), memory, and associated circuitry. It interprets and executes instructions.

program.
Executable code, data, and one or more processes. The code and data can be unique to the program or shared with other programs. A program is created by translating source programs into object modules. The Linker then links the modules together, creating a run file stored on disk. When requested by a currently active program, the operating system reads the run file into the application partition, relocates intersegment references, and schedules it for execution. The new run file can coexist with or replace other run files.

prompt.
A message that guides you through certain procedures. For example, when you use the FINISH command to finish a session, a prompt appears that tells you to confirm the command.

protected mode.
A processor mode in which application programs can access all available free memory above the first megabyte up to the maximum allowed by the processor and the hardware. See also general protection fault.

pseudotarget.
A target name that allows you to build groups of files or execute a group of commands.

PUBDEF record.
The PUBlic DEFinition record that CTOS Microsoft C emits to the object file for an initialized global variable.

public procedure.
A procedure that can be referenced from a module other than the defining module.

public installation.
An Installation Manager option that installs software on the server. Local file systems can later install the software from the server. In addition, users can delete some system files from the local disk and transparently access the corresponding publicly installed files on the server.

public symbol.
An identifier associated with a public variable, a public value, or a public procedure.

public value.
A value that can be referenced by a module other than the defining module.

public variable.
A variable that has a public address. Thus, a module other than the defining module can reference the address.

Q

QIC tape.
See quarter-inch cartridge tape.

quarter-inch cartridge (QIC) tape.
A magnetic tape storage medium that uses quarter-inch-wide tape contained in a hard plastic cartridge.

R

RAM.
An acronym for Random Access Memory.

random access memory (RAM).
The onboard processor memory where data can be written and read nonsequentially.

real mode.
A processor mode in which application programs can access memory within the first megabyte only.

reboot.
The action that resets hardware, and then reloads and executes the operating system. The operating system reboots automatically when a workstation is reset or turned on after a power failure, and when executing the BOOTSTRAP command.

release medium.
The floppy disk on which Unisys release software is distributed.

relocation.
The CTOS Loader relocates a task image in available memory by supplying physical addresses for the logical addresses in the run file.

relocation directory.
An array of locators that the CTOS Loader uses to relocate the task image.

request.
A request by a client for an operation to be performed by the operating system or by a system service process.

request-based.
An operation that uses the request procedural interface. See also request procedural interface.

request block.
A block of memory, provided by a client, which contains a special type of message that is formatted according to specific conventions, and is used by all Inter-Process Communications to the operating system. An application program (or the operating system) sends a request block to the operating system to request that a particular operation be performed.

request code.
A 16-bit value that uniquely identifies a system service. The request code routes a request to the appropriate system service process, and specifies the currently requested process.

request file.
See loadable request file.

request procedural interface.
A routine within the operating system that is executed when a program calls a request-based operation. The routine builds a request block message and calls the Request primitive, while the calling program is placed in the waiting state at its default response exchange for the system service to respond. The request procedural interface is compatible with high-level languages such as C and Pascal, as well as assembly language. See also request and request-based.

RS-422.
A high-speed communications standard for interfacing serial data transmission between peripherals, systems, and modems. The shared resource processor and server workstations use the RS-422 interface to link cluster workstations.

.run.
The standard file name suffix for run files.

run file.
An executable CTOS program. It is created by the Linker and contains object modules linked together into code and data segments.

run statement.
A line of text that you can include in a JCL file for execution by Batch on a processor.

S

sector.
The smallest addressable portion of a track on a hard or floppy disk. It contains 512 bytes.

segment.
A contiguous area of memory that contains code or data.

segmented address.
An address that specifies both a segment base and an offset.

segment registers.
This manual refers to four segment registers: the Code Segment (CS) register, the Data Segment (DS) register, the Extra Segment (ES) register, and the Stack Segment (SS) register.

sequence point.
A point in the syntax of the C language at which all side effects of an expression or series of expressions have been completed. The statement operator (;) is such a point.

server.
A workstation or shared resource processor to which cluster workstations are attached. The server provides file system, queue management facility, and other services to all cluster workstations.

server workstation.
A server that supports its own interactive programs.

shared resource processor.
A multiprocessor that permits workstations in a cluster environment to share resources. It consists of various combinations of the GP, CP, DP, FP, SP, and TP processor boards. Each processor board contains a Kernel and memory, and generally provides a subset of the services offered by a workstation operating system. Services provided by the individual processor boards can be shared among all others. See also XE 520 and XE 530.

shift operator.
An operator that shifts the left bits of the integral quantity to the left (<<) or right (>>) by the amount given in the right-hand operand. A shift operator must have operands of integral type.

short-lived memory.
The memory area in an application partition. When CTOS loads a task, it allocates short-lived memory to contain the task code and data. A client process can also load short-lived memory in its own partition.

sign extension.
The propagation of the sign bit to fill unoccupied space when promoting to a more significant type, or when performing bitwise right-shift operations.

size.
The number of bytes a data item or structure contains, unless the context indicates a different unit of measurement.

source code.
The text of a computer program before it is compiled and linked to form an executable program.

stack.
A region of program memory used for temporary data and addressed using the stack segment (SS) and stack pointer (SP) registers.

stack frame.
A region of a stack corresponding to the dynamic invocation of a procedure. It consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

stack pointer.
(SP) Indicates the top of a stack. The stack pointer is stored in the registers SS:SP.

Standard Software.
A set of programs, configuration files, and commands that are required to configure the system and perform basic operations.

statement
An executable instruction. Each statement is executed in order within a statement list unless otherwise directed by control-flow statements.

status code.
A decimal or hexadecimal number that reports the success or failure of the requested operation. A system service process stores a status code in a request block, where the client process examines it. A nonzero status usually indicates an error condition. Status code is also referred to as error code and erc.

style ID.
A six-character alphanumeric identifier for a specific Unisys product. For example, B39-CPU identifies a B39 workstation.

submit facility.
A utility that interprets a sequence of characters from a file as though they were characters typed at the keyboard. Submit files allow the convenient repetition of command sequences. See also submit file.

submit file.
A text file used by the system input process. It contains the same sequence of characters that would be typed from the keyboard. It can also contain conditional branching, variables, and video controls. When applications or requests activate submit files, file characters are returned in place of keyboard characters. See also system input process.

subparameter.
A user-supplied string in a parameter field of an Executive command form. You separate subparameters with a space.

.Sym.
.Sym is the standard file name suffix for the CTOS Linker's symbol file.

symbol.
An alphanumeric or other character, such as underscore, period, or dollar sign.

symbol file.
The Linker symbol file (suffix .Sym) that contains a list of all public symbols.

synchronous operation.
A protocol that requires a response at an exact time. Also a special type of synchronization using the CTOS request mechanism. For example, a program can issue the Wait primitive to wait at an exchange so it can synchronize its operation with a system service's response.

Sys.
An abbreviation for System. It is used in file names to denote files that are necessary for the workstation or shared resource processor to boot and function properly. It is also used as the name for the volume and directory that contain the operating system configuration.

system administrator.
The person responsible for planning, generating, extending, and controlling the use of the operating system and system services to improve the overall productivity of the installation.

system crash.
An abnormal condition from which the system cannot recover. After a crash, the system usually freezes or reboots automatically.

system disk.
The system disk is the disk on which the workstation server or shared resource processor system software is installed.

system error log.
A file containing information about many types of hardware and software errors. You can view or print it with the PLog command.

system input process.
A CTOS process that permits a sequence of characters from a submit file to be interpreted as though they were characters typed at the keyboard. Submit files allow the convenient automation of command sequences. See also submit facility and submit file.

system service.
A program or subprogram that expands the capabilities of the operating system. It manages or provides access to a resource. A resource may be hardware, such as a mouse or printer, or other system services.

T

target.
Listed in a makefile, a file that you want to create, such as an executable file.

task.
Executable code, data, and one or more processes.

task image.
A program stored in a run file that contains code segments and/or static data segments.

text file.
A file that contains bytes that represent printable characters or control characters (such as tab, newline, and so on).

U

user configuration file.
A file containing a user profile. See also user file.

user file.
A file in the system directory that identifies the user and specifies the environment the system activates after the user signs on.

user name.
The name a user signs on with. The file name of a user file, in the form [Sys]<Sys>UserName.user, defines the user name.

V

version 4 run file
A real-mode run file, one that cannot run above the 1 mb barrier. Version 4 run files are not compatible with protected mode.

version 6 run file
A protected-mode run file, one that can run above the 1 mb barrier.

W

wild card character.
A character that represents one or more unspecified characters. CTOS has two wild card characters: an asterisk (*) and a question mark (?). The asterisk represents any string of characters; the question mark represents individual characters. For some operations, you can use wild card characters in file specifications. The system then tries to match the portion of the name that appears before or after the wild card character and performs the requested operation for each matched file name. For example, *.ind would expand to all files that end in .ind.

X

XE 520.
A real-mode server multiprocessor system that permits workstations in a cluster environment to share resources. It consists of various combinations of the CP, DP, FP, SP, and TP processor boards. Each processor board contains a Kernel and memory, and generally provides a subset of the services offered by a workstation operating system. Services provided by the individual processor boards can be shared among all the others.

XE 530.
A protected-mode server multiprocessor system that permits workstations in a cluster environment to share resources. It consists of various combinations of the GP, CP, DP, FP, SP, and TP processor boards. Each processor board contains a Kernel and memory, and generally provides a subset of the services offered by a workstation operating system.

Index

\ par, 11-16
\f escape sequence, 10-7
\g escape sequence, 10-7
\n escape sequence, 10-7
_asm blocks
  defining _asm blocks as C, 8-18
  referring to C variables, 8-11
  tags not recognized in, 8-10
  using assembly language in, 8-4
_asm keyword
  defined, 8-2
_beginthread function, 16-6
_cdecl keyword, 6-29
_cdecl prototypes, 5-36
_ctoserrno, A-5
_endthread function, 16-7
_export keyword, 16-18
_fastcall, 6-29
_fastcall calling convention
  argument-passing convention, 6-31
  function-naming convention, 6-33
  register preservation requirement, 6-32
  stack adjustment convention, 6-32
_fortran keyword
  using, 14-13
_huge memory support for real mode, 5-21
_loadds keyword, 16-19
_pascal keyword, 6-30
  using, 14-13
_plm keyword, 6-30
  limitations, 5-37
  using, 5-36
  what it does, 14-14
!CMDSWITCHES directive, 10-23
!ELSE directive, 10-23
!ENDIF directive, 10-23
!ERROR directive, 10-23
!IF directive, 10-23
!IFDEF directive, 10-24
!IFNDEF directive, 10-24
!INCLUDE directive, 10-24
!UNDEF directive, 10-24
.category dot command, 11-24
.command dot command, 11-24
.comment dot command, 11-24
.context dot command, 11-24
.end dot command, 11-24
.freeze dot command, 11-24
.IGNORE pseudotarget, 10-27
.length dot command, 11-24
.list dot command, 11-25
.mark dot command, 11-25
.next dot command, 11-25
.paste dot command, 11-25
.popup dot command, 11-25
.previous dot command, 11-25
.ref dot command, 11-25
.SILENT pseudotarget, 10-27
.SUFFIXES pseudotarget, 10-27
.topic dot command, 11-25
/FPa floating-point option, 9-13
/FPc floating-point option, 9-11
/FPc87 floating-point option, 9-12
/FPi floating-point compiler option, 9-10
/FPi87 floating-point option, 9-11

/G0 optimization option, 6-21
/G1 optimization option, 6-21
/G2 optimization option, 6-21
/Gc optimization option, 6-30
/Gd optimization option, 6-29
/Gr optimization option, 6-30
/Gs optimization option, 6-18
/Gt optimization option, 7-26
/ND optimization option, 7-27
/Oa optimization option, 6-12
/Oc optimization option, 6-20
/Od optimization option, 6-5
/Oe optimization option, 6-18
/Og optimization option, 6-20
/Oi optimization option, 6-8
/Ol optimization option, 6-15
/On optimization option, 6-16
/Op optimization option, 6-20
/Os optimization option, 6-8
/Ot optimization option, 6-8
/Ow optimization option, 6-12
/Ox optimization option, 6-22
/Oz optimization option, 6-16
80186 processor, 6-21
80188 processor
  instruction set, 6-21
  operating systems, 6-22
  using, 6-21
80286 processor, 6-21
80x87 coprocessors
  register size, 6-20
87.LIB
  defined, 5-23

A

abort function, C-27
Accessing parameters in
  assembly, 14-27
Addition operator (+), 10-25
Address space
  portability, 15-19
Address variables
  mixed-language
    programming, 14-41
Addresses
  passing with common blocks
    in mixed languages, 14-42
Addressing
  based, 7-5
ADR type, 14-41
ADRMEM, 14-41
ADS type, 14-41
ADSMEM, 14-41
Advisor, 4-1
Advisor Help library APIs,
  11-34
Aggressive optimizations
  enabling, 6-16
Alias, 6-12
Aliasing
  assuming no, 6-12
  rules for nonvolatile
    variables, 6-12
  troubleshooting, 6-14
Allocating global registers, 6-18
Allocating local data in
  assembly, 14-24
ALT key
  maps to COPY, 3-2, 12-1
Alternate math libraries
  xLIBFA.LIB, 5-24
Alternate math package, 9-6
Ampersand (&), 14-41
ANSI standards and CTOS
  implementation
  arguments to main(), C-3
  arrays, C-11
  availability of registers, C-12
  bits per character, C-5
  case distinction, C-4
  casting integers to
    floating-point values, C-10
  casting pointers, C-11
  character sequences, C-18
  character sets, C-5
  character testing, C-21
  characters, C-5

ANSI standards and CTOS
implementation (continued)
converting multibyte characters, C-7
declarators, C-15
demotion of integer values, C-8
diagnostics, C-2
environment, C-2
environment names, C-28
floating-point math, C-10
identifiers, C-4
integers, C-7
interactive devices, C-3
largest array size, C-11
library functions, C-19
multibyte characters, C-5
pointer subtraction, C-12
pointers, C-11
pragmas, C-18
preprocessing directives, C-16
qualifiers, C-15
range of char values, C-7
range of integer values, C-8
registers, C-12
remainders, C-9
right shifts, C-9
signed bitwise operations, C-9
significant characters without external linkage, C-4
statements, C-15
structures, C-13
translation, C-2
truncation of floating-point values, C-11
unions, C-13
unrepresented character constants, C-6
wide characters, C-6
Append mode
file position in, C-23
Applications
building, 5-1, 16-1
argData, 12-21
Argument types and register candidates, 6-31
Arithmetic
one's complement, 15-14
pointer arithmetic
huge versus far, 7-4
pointer in huge models, 7-9
two's complement, 15-14
Arithmetic mode
processor, 15-14
Arrays
ANSI standards and CTOS implementation, C-11
automatic void with huge, 7-8
declaring in mixed languages, 14-38
displaying in CodeView, 13-21
equivalent declarations among languages, 14-39
indexing in mixed languages, 14-38
limits on huge, 7-8
names always passed as pointers, 14-13
static and huge, 7-8
using with int data types, 15-4
Assembly
accessing parameters, 14-27
allocating local data, 14-24
C calls to, 14-21
entering procedures, 14-24
exiting procedures, 14-31
preserving register values, 14-25

Assembly (continued)
returning values, 14-30
setting up procedures, 14-23
standard interface method, 14-23
writing procedures, 14-22
assert function
diagnostic printed by, C-21
astcall calling convention
return value convention, 6-32
Asterisk (*), 10-7
At sign (@), 6-33, 11-16
atexit function, C-27
Availability of registers
ANSI standards and CTOS implementation, C-12

B

Back command button, 4-17
Backslash (\), 10-6
Based addressing, 7-5
Based variables, 7-30
Binary mode, 5-41
Binary streams
number of null characters that can be appended, C-23
Bit fields
alignment, C-14
alignment of data defined in, 15-14
in structures, 15-12
order of assignment in structures, 15-13
size and alignment in structures, 15-13
storage, C-14
Bits per character
ANSI standard and CTOS implementation, C-5
Bitwise AND operator (&), 10-25
Bitwise operations
AND (&), 10-25
OR (|), 10-25
right shift, 15-24
signed, C-9
XOR (^^), 10-25
Bitwise OR operator (|), 10-25
Bitwise XOR operator (^^), 10-25
Blank lines, C-23
BP as frame pointer, 14-24
Braces ({ })
enclosing an _asm block, 8-3
enclosing text Choice symbol (|), viii
Brackets
square ([ ]), 10-26
Bracketted file names
including, C-16
Bracketted text brackets ([]), viii
Breakpoints
displaying in CodeView, 8-7
selecting and editing in CodeView, 13-13
setting and removing in CodeView, 13-13
setting values in CodeView, 13-15
using in CodeView, 13-17
Browser
building files for the, 5-11
creating a debug build, 3-55
debugging with the, 3-57
overview, 1-7
starting, 3-13
using, 3-55, 3-59

BTOS C functions
documented but not supported, A-3
summary of status in CTOS Microsoft C, A-22
BTOS C math library
summary of function locations in CTOS Microsoft C, A-30
Buffering files, C-23
Building applications, 5-1
Building DLLs with Microsoft C, 16-21
Building run files, 5-1, 3-31
Buttons
command, 4-17
Bytes
CTOS Microsoft C ordering, 15-31
order in a word, 15-10
ordering for long types on various machines, 15-32
ordering for short types on various machines, 15-32

C

C
data type naming conventions, 14-34
using, 8-8
C calling convention, 6-29
C calls
to assembly language, 14-21
to FORTRAN, 14-15
to high-level languages, 14-12
to Pascal and PL/M, 14-18
C functions
_asm blocks to C functions, 8-17
migrating to CTOS Microsoft C, A-2
run time library functions for thread control, 16-5
C run-time library support of long double types, 9-5
C stack frame example, 14-28
C string format, 14-35
C symbols
using in _asm blocks, 8-9
Call to math coprocessor compiler option, 9-12
Calling a FORTRAN function from C, 14-17
Calling a FORTRAN subroutine from C, 14-15
Calling a function, 6-25
Calling a Pascal or PL/M function from C, 14-20
Calling a Pascal or PL/M procedure from C, 14-19
Calling conventions
_fastcall, 6-31
affects programming three ways, 14-7
Assembly-language and C, 14-22
calloc function, C-27
defaults for languages, 14-33
effects of, 14-8
FORTRAN and Pascal compared, 14-7
mixed-language programming, 14-7
stack adjustment, 6-32

Calls
C to FORTRAN and Pascal, 14-12
from C to FORTRAN subroutines, 14-15
mixed-language protocol, 14-7
steps for executing a mixed-language call from C, 14-12
to Assembly, 14-21
to emulator compiler option, 9-11
to programs written in other languages, 14-1
to routines in different languages, 14-3
Caret (^), 10-9
Case distinction
ANSI standard and CTOS implementation, C-4
in NMAKE macros, 10-15
Case sensitivity
portability issues, 15-26
Case translation
migration issues, A-7
portability issues, 15-21
Casting
pointer portability issues, 15-16
Casting integers to floating-point values
ANSI standard and CTOS implementation, C-10
Casting pointers
ANSI standards and CTOS implementation, C-11
Char data type
range of values, C-7
size, 15-3
Character
ANSI standard and CTOS implementation, C-5
bits per, C-5
Character classification
portability issues, 15-20
Character constants, C-6, C-16
Character sequences
ANSI standards and CTOS implementation, C-18
Character sets
ANSI standard and CTOS implementation, C-5
portability, 15-20
portability issues, 15-30
portability restrictions, 15-20
source and execution, C-6
Character testing
ANSI standard and CTOS implementation, C-21
Characters
case sensitivity and CL command line options, C-4
converting multibyte characters, C-7
multibyte, C-5
number of significant without external linkage, C-4
significance of case distinction, C-4
terminating newline, C-22
wide, C-6
CHAR_BIT, 15-4
CHAR_MAX, 15-4
CHAR_MIN, 15-4
Check_stack pragma, 6-18
CL
automatically invoking MASM, 5-3
command form, 5-2
command form summary, B-1

CL (continued)
command line option notes, 5-3
command line options, B-1
command line syntax, 5-2
controlling automatic linking, 5-9
default memory model, 5-3
global register allocation option, 15-26
how to use, 5-2
overlay switches, 5-4
overview, 1-2
sample command form that creates a run file, 5-10
sample command form with linker options, 5-21
syntax errors, 5-5
CLD instruction, 5-36
Clock cycles of processor for calling sequence, 6-27
clock function, C-30
cmdTable, 12-16
CodeView
advanced techniques, 13-29
calling functions, 13-30
changing data, 13-28
changing register values, 13-28
changing variables, 13-20
checking for undefined pointers, 13-30
command form summary, B-10
configuring, 2-14
conflict with CTOS Librarian, 3-63, 13-3
controlling program execution, 13-12
creating debug builds with MCBIND, 13-4
CodeView (continued)
creating debug builds with MLINK, 13-4
customizing from TOOLS.INI, 13-33
dialog command size specifiers, B-18
displaying arrays, 13-21
displaying expressions in the Watch window, 13-20
displaying memory, 13-25
displaying processor registers, 13-27
displaying structures, 13-21
displaying variables in the Watch window, 13-19
displaying in CodeView, 13-26
executing programs continuously, 13-12
format specifiers, B-18
handling register variables, 13-32
modifying variables, registers, and memory, 13-27
opening multiple source windows, 13-30
overview, 1-6
preparing a debug build, 3-62, 13-2
print from, 13-31
redirecting input and output, 13-32
replaying a debug session, 13-28
Run and Animate commands, 13-18
selecting and editing breakpoints, 13-13

CodeView (continued)
setting and removing breakpoints, 13-13
setting breakpoint values, 13-15
setting command-line arguments, 13-29
single stepping through code, 13-18
starting, 13-6
starting and running from PWB, 3-64
Step and Trace commands, 13-18
system configuration and starting, 13-7
using breakpoints, 13-17
using History On, 13-29
using the Quick Watch window, 13-24
viewing and modifying program data, 13-19
CodeView windows
adjusting, 13-11
entering commands in, 13-10
identifying active, 13-10
list of, 13-8
opening manually and automatically, 13-9
selecting, 13-10
Coding conventions
migration issues, A-14
Combined libraries
creating and naming, 5-25
creating with MLIB, 5-26
xLIBC7.LIB, 5-25
xLIBCA., 5-25
xLIBCE.LIB, 5-25
COMDEF OMF record
versus PUBDEF record, A-20
Command buttons, 4-17
Command form summaries
CL, B-1
CodeView, B-10
Environment Service, B-19
ML, B-29
MLIB, B-28
NMAKE, B-38
PWB, B-40
PWBRMAKE, B-41
QuickHelp, B-43
Command forms
CV, 13-7
CV Run, 13-7
HelpMake, 11-5
MCBIND, 5-20
MLIB, 5-26
NMAKE, 10-3
QuickHelp, 4-2
Command line
controlling optimization from, 6-3
Command line options
CL, B-1
HelpMake, 11-6
MLIB, B-28
MLINK, B-29
NMAKE, 10-28, B-38
PWBRMAKE, B-41
QuickHelp, B-43
Command lines
indentation of NMAKE, 10-6
Commands
.context, 11-2, 11-15
Browser, 3-59
CL, 5-2
CodeView, B-12
CV, 13-7
CV RUN, 13-7
HelpMake syntax, 11-5
MLINK, 5-7
NMAKE, 10-3

Commands (continued)
QuickHelp, 4-2
QuickHelp dot, 11-23
Comments
in a description file, 10-11
in-line assembly, 8-19
nested, 10-11, A-19
using old-style C in _asm blocks, 8-18
Common blocks
mixed-language programming, 14-42
Common subexpression elimination, 6-5
Common subexpression optimization, 6-20
Compile options, 16-19
Compiler
how to use CL, 5-2
inserting leading underscores with _plm and _pascal keywords, 14-5
limits, D-2
limits and numerical ranges, D-1, D-4
naming conventions, 14-4
numerical ranges for constants defined in LIMITS.H, D-4
numerical values for constants defined in FLOAT.H, D-6
optimization, 6-1
options and migration issues, A-15
program limits at run time, D-3
three methods for passing parameters, 14-9
Compiler assumptions
program portability, 15-22
Compiler options
/Ad, 7-23
/Au, 7-24
/Aw, 7-24
/GY disabling /Oc, 5-4
/GY incompatibility with intrinsic forms of mem* and str*, 5-5
/r disabling /Oc, 5-4
/r incompatibility with /Oe and /Og, 5-4
calls to emulator, 9-11
calls to math coprocessor, 9-12
choosing for memory models, 7-7
in-line emulator, 9-10
in-line math coprocessor, 9-11
migration issues, A-15
setting in PWB, 3-35
summary of BTOS C and CTOS Microsoft C, A-16
use alternate math, 9-13
used for the Browser, 3-13
used to stop automatic linking with MLINK, 5-3
using to migrate huge memory models, A-11
Compiling
choosing when compiling mixed languages, 14-11
FORTRAN routines with the $TRUNCATE metacommand enabled, 14-5
mixed-language programming, 14-11
overview, 1-2
PWB extensions, 12-29

Compiling (continued)
with different languages, 14-1
COMPLEX*16 type, 14-35
COMPLEX*8 type, 14-35
Configuration file
Environment Service, 2-12
linker, 6-24
Constant propagation, 6-6
Constants
character, C-16
unrepresented character, C-6
Contents command button, 4-17
Context Manager
creating a QuickHelp partition, 4-25
Context prefixes, 11-17
Contexts
defined, 11-2
local, 11-16
Control characters
using as literals, 10-8
Conventions
manual, viii
mixed-language programming, 14-4
Pascal, 14-33
Converting header files, A-7
CPU registers
use of, 6-18
Creating run files
sample CL command form, 5-10
Cross references
defined, 11-2
hyperlinks, 11-2
implicit, 11-2
maps to CODE, 3-2, 12-1
QuickHelp, 11-27
CTOS III Applications
building, 16-1
CTOS Microsoft C
new features, 1-11
overview, 1-1
CTOS Microsoft Help File Creation Utility (HelpMake), 11-1
CTOS Microsoft Macro Assembler (MASM)
advantages over in-line assembler, 14-21
versus using in-line assembly, 8-1
CTOS.Lib, 2-14
CTOS.Lib.h, 5-36, A-7
CTOSToolKit.Lib, 2-14
CURRENT.STS, 3-2, 3-31, 13-9
Customizing your
optimizations, 6-7
CV
command form, 13-7
CV RUN
command form, B-10

D

Dash (-), 11-6, C-25
Data
handling in mixed-language programming, 14-32
Data files
portability issues, 15-30
Data ranges defined in LIMITS.H, D-4
Data threshold
setting, 7-26
Data type
char sizing, 15-3
declaring floating-point, 9-1
int sizing, 15-3
naming conventions for C, FORTRAN, and Pascal,

Data type (continued)
14-33
order and alignment issues with structures and unions, 15-7
short int sizing, 15-3
size of basic, 15-2
variable size, 8-6
Date
default, C-19
Directives
data and operations, 8-5
even and align, 8-5
supported by in-line assembler, 8-5
DBL_DIG, 15-6
DBL_DIG, D-6
DBL_MAX, 15-6
DBL_MAX, 15-6
DBL_MAX_10_EXP, 15-6
DBL_MAX_EXP, 15-6
DBL_MIN, D-6
DBL_MIN_10_EXP, 15-6
DBL_MIN_EXP, 15-6
Dead-store elimination, 6-6
Debug builds
advanced CodeView techniques, 13-29
creating Browser, 3-13
creating for CodeView, 13-2
from PWB, 3-53
overview of CodeView techniques, 13-11
problems to avoid in assembly, 8-7
with error messages in PWB, 3-54
with the Browser, 3-13, 3-55, 3-57
Declarators
ANSI standards and CTOS implementation, C-15
Decoding Help databases
decoded format Help files 11-4
Decoding options
HelpMake, 11-11
Default optimizations, 6-5
Demotion of integers
ANSI standard and CTOS implementation, C-8
Denormalized numbers
extending range of floating-point numbers with, 9-3
Dependency line
NMAKE, 10-4
Dependency tree, 10-6
Dependents
defined, 10-2
Description blocks
control characters as literals, 10-8
dependency tree, 10-6
dependents, 10-5
escape sequences, 10-7
example, 10-5
extension search paths, 10-21
indentation, 10-6
NMAKE, 10-4
pseudotargets, 10-27
specifying a target in multiple, 10-10
targets, 10-5
wildcards, 10-7
Description command button, 4-22
Description files
comments, 10-11
defined, 10-2
dependents, 10-5
description block example, 10-5

Description files (continued)
description blocks, 10-4
directives, 10-23
elements, 10-4
escape sequences, 10-7
example, creating a library, G-21
example, creating a library version string, G-4
example, creating a PWB extension, G-6
example, creating a run file, G-12
example, creating run files, libraries, and a system service, G-30
extension search paths, 10-21
how NMAKE interprets, 10-31
listing targets in multiple description blocks, 10-10
macros, 10-11
NMAKE examples, G-1
placing newline characters in, 10-9
targets, 10-5
user defined inference rules, 10-20
using control characters as literals, 10-8
wildcard characters, 10-7
Designing DLLs
designing and writing, 16-13
DGroup
controlling placement of near heap memory, 5-27
placing near heap memory in, 5-28
Diagnostics
ANSI standard and CTOS implementation, C-2
Dialog boxes, 4-15
Dialog commands
CodeView, B-14
Directives
!CMDSWITCHES, 10-23
!ELSE, 10-23
!ERROR, 10-23
!IF, 10-23
!IFDEF, 10-24
!IFNDEF, 10-24
!INCLUDE, 10-24
!UNDEF, 10-24
DB, 8-7
EXTERN and assembly, 14-23
for NMAKE description files, 10-23
NMAKE, 10-23
PUBLIC and assembly, 14-23
defaults, 2-5
Directory name
size, C-24
Disk space used by an executable file
reducing, 6-24
Division operator (/), 10-25
DLLs
and CTOS Microsoft C libraries, 16-10
application programs, 16-10
linking files needed, 16-26
with static C run-time library functions, 16-21
without C run time library functions, 16-12

Domain errors, C-22
Dot commands, 11-23
Double colon (::), 10-10
Double dollar sign ($, 10-14
Double float data type
  size, 15-5
Double float data type
  support, 9-1
DS allocation
  migration, A-21
Dynamic linking with CTOS III
  DLL, 16-8

E

EBADF constant, C-26
EDOM error constanty, C-22
Effects of calling conventions,
  14-8
EINVAL constant, C-26
Ellipsis, viii
EM.LIB
  defined, 5-23
Emulator package, 9-5
Encoding options, 11-7
End-of-file escape sequence,
  10-7
ENOENT constant, C-28
Entering the assembly
  procedure, 14-24
Envelope function
  migrating to 6.1, 5-36
Environment
  ANSI standard and CTOS
    implementation, C-2
Environment differences
  program portability issues,
    15-30
Environment names
  ANSI standard and CTOS
    implementation, C-28
Environment Service
  command form summary,
    B-19
  commands, B-20
  disabling CL and MLINK
    variables for PWB, 3-2
  disabling CL and MLINK
    variables for PWB, 5-34
  installing, 2-12
  overview, 1-11
  using, 5-33
  variables, examples, and
    associated commands, B-21
Environment variables
  CL description, B-22
  descriptions, B-22
  HELPFILES description,
    B-22
  INCLUDE, C-16
  INCLUDE description, B-22
  INIT description, B-22
  LIB description, B-22
  M, B-22
  MAKEFLAGS desc, B-22
  MLINK, 5-11
  NO87, 9-15
  PATH description, B-22
  QH description, B-22
  recognized by QuickHelp,
    B-43
  recognized by CL, B-2
  recognized by CodeView,
    B-10
  recognized by HelpMake,
    B-23
  recognized by MLIB, B-28
  recognized by MLINK, B-30
  recognized by NMAKE, B-38
  recognized by P, B-40
  TMP description, B-22

EnvironmentConfig.Sys
  sample, 2-12
  syntax rules, 5-33
Equality operator ( ), 10-25
ERANGE macro, C-22
errno, A-5
errno values
  supported, 5-41
Error information
  accessing in QuickHelp, 4-23
Error messages
  errno values supported, 5-41
  interpreting in PWB, 3-54
  manipulating, 5-40
Errors
  checking returns from
    dynamic memory allocation
    routines, 15-19
  domain, C-22
  file position, C-26
  message syntax, C-2
Escape sequences
  \f, \g, and \n for NMAKE
    description files, 10-7
  NMAKE, 10-7
EVBackgroundOff
  Config.Sys entry to make
    VGA work, 2-14
EVEN Directives
  ALIGN, 8-5
  CodeView commands, B-12
  dialog commands, B-14
Example command button,
  4-22
Exclamation point (!), 10-23
Executable file
  reducing disk space used,
    6-24
Exiting assembly procedures,
  14-31
Exiting QuickHelp, 4-4
Explicit cross references, 11-27
Exponents
  bias, 9-3
  length of for floating-point
    data types, 9-2
  storage, 9-3
Expressions
  displaying in CodeView,
    13-20
  evaluation order and
    portability, 15-27
  SIZE, 8-9
Extended keywords
  summary of BTOS C and
    CTOS Microsoft C, A-13
Extension search paths
  NMAKE, 10-21
Extensions
  PWB extensions, 12-13
External data
  mixed-language
    programming, 14-40

F

facility, A-21
Far call optimization
  enabling, 6-25
Far heap memory
  allocating, 5-31
  sizing and allocating, 5-30
Far pointers
  how they work, 7-3
FarCallTranslation, 6-25
  benefits, 6-26
  how it affects code, 6-26
fdtofh function, A-4
fgetpos function, C-26
File access limits, C-24
File buffering, C-23
File compression, 11-7

File handles
increasing the number of, 2-15
File names
characters permitted, C-24
including bracketted, C-16
quoted, C-17
size, C-24
File position errors, C-26
File streams, A-5
Files
deleting open, C-24
increasing the maximum number of open, 2-15
increasing the number of handles, 2-15
increasing the number of streams, 2-16
zero-length, C-24
Flavor object modules
defined, 5-21
MCMEMORY.Obj, 5-32
MIN.Obj, 5-22
REALHUGE.Obj, 5-22
Float data type
portability assumptions, 15-5
size, 15-5
support for, 9-1
FLOAT.H
numerical values defined in, D-6
numerical values for defined constants, D-6
Floating-point data types
declaring, 9-1
declaring functions that return, 9-4
declaring variables as, 9-2
differences between, 9-2
range, 9-3
Floating-point math
alternate math package, 9-6
ANSI standard and CTOS implementation, C-10
compatibility between compiler options, 9-14
compiling and linking options, 9-7
controlling operations, 9-1
coprocessor package, 9-6
emulator package, 9-5
library considerations, 9-13
packages, 9-5
required for DLLs, 10-13
selecting operations, 9-7
summary of compiler options, 9-8
Floating-point math libraries
87.LIB, 9-5
EM.LIB, 9-5
xLIBFA.LIB, 9-11
xLIBFP.LIB, 9-9
Floating-point numbers
extending range of with denormalized format, 9-3
Floating-point results
achieving consistent, 6-20
Floating-point routines
accessing, 9-7
Floating-point values
casting integers to, C-10
effect of /Op on, 6-21
minimun and maximum, 9-3
truncation, C-11
underflow, C-22
Floating-point variables
precision and storage compared, 9-4
promoting, 9-4

FLT_DIG, 15-6
FLT_DIG, D-6
FLT_MAX, 15-6
FLT_MAX, D-7
FLT_MAX_10_EXP, 15-6
FLT_MAX_EXP, 15-6
FLT_MIN, 15-6
FLT_MIN, D-7
FLT_MIN_10_EXP, 15-6
FLT_MIN_EXP, 15-7
fMeta, 12-21
fmod function, C-22
Format specifiers
  printf, E-1
  scanf, E-1
Formatting flags
  QuickHelp, 11-26
FORTRAN
  accessing common blocks directly, 14-43
  C calls to, 14-15
  calling a subroutine from C, 14-15
  calling conventions, 14-7
  data type naming conventions, 14-34
  indexing arrays, 14-38
  naming conventions, 14-4
  routines compiled with the $TRUNCATE metacommand enabled, 14-5
  string format, 14-36
  variable length strings not supported, 14-36
FORTRAN/Pascal calling convention, 6-30
fprintf function, C-25
Frame pointer, 14-24
ftell function, C-26
Function
  not supported, 5-42
Function calls
  restrictions on using intrinsic forms, 6-11
Functions
  _fastcall, 6-32
  abort, C-27
  assert, C-21
  atexit, C-27
  BTOS C locations in CTOS header files, A-22
  calling FORTRAN from C, 14-17
  calling from CodeView, 13-30
  calling two ways, 6-25
  character limits, A-15
  clock, C-30
  documented in BTOS C but not supported by CTOS Microsoft C, A-3
  documented in BTOS C with functional substitutes in CTOS Microsoft C, A-4
  envelope, 5-36
  fgetpos, C-26
  floating-point without true intrinsic forms, 6-10
  fmod, C-22
  fprintf, C-25
  ftell, C-26
  generating intrinsic, 6-8
  intrinsic forms of mem* and str*, 5-5
  intrinsic forms used when compiling with /Oi, 6-9
  isalnum, C-21
  isalnum, C-21
  iscntrl, C-21
  islower, C-21
  isprint, C-22

Functions (continued)
isupper, C-22
longjmp, 5-3
messages generated by perror, C-26
migrating to CTOS Microsoft C, A-2
migration issues, A-22
miscellaneous bahaviors, 5-42
perror, 5-40
portability issues, 15-28
portable run-time, 15-20
prototypes and migration, A-14
prototyping, 8-9
rename, C-25
returning values to assembly code, 8-12
Run-time library functions
re-entrant, F-1
rules for declaring near and far, 7-18
setjmp, 5-3
strerror, C-28
summary of BTOS C status, A-22
system, C-28
that return floating-point types, 9-4
undocumented BTOS C with work-arounds, A-5
with a variable number of arguments, 15-27
Function-calling conventions, 6-29

G

GDT protected run files
creating with MLINK, 5-25
Global register allocation, 6-18
Global register allocation CL option, 15-26
GO escape sequence, 10-7
Greater than operator (>), 10-25

H

H. contexts, 11-17
Hardware assumptions
program portability, 15-2
Header files
converting for migration, A-7
correlating BTOS C and CTOS Microsoft C, A-8
migration, A-6
Help
accessing error information, 4-23
copying and pasting, 4-18
dialog boxes, 4-15
displaying keyword information, 4-21
navigating through screens, 4-15
printing, 4-24
structure, 4-5
using, 4-1
using hyperlinks, 4-17
using pull-down menus and dialog boxes, 4-7

Help database
content and format, 11-1
creating a, 11-13
using formats, 11-21
Help delimiter (>>), 11-16
Help file formats
ASCII, 11-22
minimally formatted ASCII, 11-4
QuickHelp, 11-4
QuickHelp components, 11-22
QuickHelp cross references, 11-27
QuickHelp dot commands, 11-23
QuickHelp formatting flags, 11-26
RTF, 11-22
supported, 11-3
Help files
ASCII format, 11-4
assumptions about format and file structure, 11-15
contents, 11-2
context prefixes, 11-17
controlling text appearance with Helpmake format flags, 11-3
creating with Helpmake, 11-1
creating with two methods, 11-13
cross references, 11-2
database content and format, 11-1
decoding, 11-5
decoding options, 11-11
encoding, 11-5
Help files (continued)
examples of how to create, 11-5
formats supported, 11-3
h. contexts, 11-17
hyperlinks, 11-19
local contexts, 11-16
manifext constants, 11-29
QuickHelp format, 11-4
RTF format, 11-22
structure of databases, 11-15
supported standard formats, 11-1
transferring between applications, 11-4
using the three text formats, 11-21
Help text file structure, 11-15
HelpMake
command examples, 11-5
command line options, 11-6
command form summary, B-23
command syntax, 11-5
creating Help files, 11-1
decoding options, 11-11
High-level languages
C calls to, 14-12
How languages push parameters onto the stack, 14-8
Huffman compression, 11-7
Huge memory model
migration, A-11
Huge pointers
how they work, 7-4
Hyperlink cross references, 11-2
Hyperlinks
command buttons, 4-17
defined, 11-3
navigating through Help with, 4-17
using, 4-17

I

Identifiers
  ANSI standard and CTOS implementation, C-4
  length, case, and portability, 15-25
  naming convention in mixed-language programming, 14-4
Implementation-defined behavior
  ANSI related, C-1
Implicit cross references, 11-2
INCLUDE environment variable, C-16
Increasing the maximum number of open files, 2-15
Increasing the number of file handles, 2-15
Increasing the number of streams, 2-16
Index command button, 4-17
Inequality operator (!), 10-25
Inference rules, 10-20
Initialization
  and termination for DLLs, 16-14
  migration issues, A-15
Input redirection, 5-39
Installation
  and configuration, 2-11
  default directories, 2-5
  Environment Service, 2-12
  OS requirements, 2-1
  preliminary operations, 2-2
  RAM requirements, 2-1
  Task Management Service, 2-11
  VGA notes, 2-14
Instruction set
  in-line assembly instruction set, 8-4
Int data type
  size, 15-3
  size assumptions and portability, 15-4
Integer values
  range, C-8
Integers
  ANSI standard and CTOS implementation, C-7
  casting to floating-point values, C-10
  demotion, C-8
Interactive devices
  ANSI standard and CTOS implementation, C-3
Intrinsic function calls
  restrictions on using, 6-11
Intrinsic functions
  generating, 6-8
  math, 6-10
Intrinsic pragma, 6-10
INT_MAX, 15-5
INT_MAX, D-5
INT_MIN, 15-5
INT_MIN, D-5
In-line assembler
  advantages of in-line, 8-2
  advantages over MASM, 14-22
  in-line code features, 8-2
  MASM advantages over, 14-21
  uses of in-line, 8-2
  writing in-line assembly, 8-12
In-line emulator compiler option, 9-10
In-line files
  NMAKE, 10-30

In-line instructions or calls, 9-13
In-line math coprocessor instructions compiler option, 9-11
isalnum function, C-21
isalpha function, C-21
iscntrl function, C-21
islower function, C-21
isprint function, C-22
isupper function, C-22
Italicized text, viii

K

Keystrokes
_asm before assembly instruction, 8-18
_based, 7-12
_cdecl, 6-29
_far addressing, 7-12
_fastcall Functions
_fortran, 14-13
_fortran and maxparam, 14-16
_huge, 7-8
_near addressing, 7-12
_near and _far with _plm, _fortran, and _pascal, 14-13
_pascal, 6-30, 14-13
_plm, 5-36, 6-30, 14-13
_plm and _pascal, 14-5
_plm effects, 14-14
_segment, 7-32
_segname, 7-32
_self, 7-32
Keystrokes (continued)
assigning in PWB, 12-4
changing in PWB, 12-4
disabling in PWB, 12-5
examples of using _plm, _pascal, and _fortran, 14-14
migration of language, A-12
mixed-language naming conventions, 14-5
nonportable, 15-31
specific to CTOS Microsoft C, 15-31
summary of BTOS C and CTOS Microsoft C extended, A-13
syntax rules and examples, 14-14
used with _based pointer declarations and statements, 7-32
using _far in mixed languages, 14-11
using _plm, _fortran, and _pascal, 14-12
public and using _fastcall with mixed languages Compiler
/Gr option and mixed languages, 14-9

L

Labels
jumping to _asm blocks
jumping to labels, 8-15
Language convention requirements, 14-4

Language equivalents for
routine calls, 14-3
Language keywords
migration, A-12
Largest array size
ANSI standards and CTOS
implementation, C-11
LDBL1_MIN, 15-7
LDBL_DIG, 15-7
LDBL_MAX, 15-7
LDBL_MAX_10_EXP, 15-7
LDBL_MAX_EXP, 15-7
LDBL_MIN_10_EXP, 15-7
LDBL_MIN_EXP, 15-7
Left shift operator (<<), 10-25
Less than operator (<), 10-25
Less than or equal to operator
(<), 10-25
LIBCEXT.LIB
defined, 5-23
Librarian
overview, 1-10
Libraries
87.LIB, 5-23
combined, 5-22
conflicts between C and
Pascal, 14-3
coprocessor, 5-23
creating and naming
combined, 5-25
CTOS, 2-14
CTOS.Lib, 5-24
default, 5-17
EM.LIB, 5-23
emulator, 5-23
LIBCEXT.LIB, 5-23
list of standard, 5-23
required for linking, 5-21
specifying standard, 5-22
unsupported functions, 5-42
xLIBCP.LIB, 5-23
xLIBFA.LIB, 5-24
xLIBFP.LIB, 5-25
Library considerations for
floating-point options, 9-13
Library functions
ANSI standards and CTOS
implementation, C-19
migration, A-22
Library support
for mulithread program, 16-2
LIMITS.H
data ranges defined in, D-4
numerical ranges for defined
constants, D-4
Line-continuation character (\),
10-6
LINK description, B-22
Linker
cross references between
MLINK and MCBIND, 5-15
how to specify options from
CL, 5-9
MLINK command form, 5-7
using, 5-6
Linker configuration file, 6-24
Linker options
setting in PWB, 3-39
Linker options that control
optimization, 6-24
Linker response files, 5-12
LinkerConfig.sys, 6-24
Linking
controlling automatic, 5-9
floating-point support, 5-25
for real mode with _huge
memory, 5-32
mixed-language
programming, 14-11
overview, 1-2
PWB extensions, 12-31
rules that void with alternate
math library, 9-11
with floating-point libraries
other than the default, 9-14

Linking minimum-sized run file, 5-24
Literals
using control characters as, 10-8
LLIBCMT.LIB, 16-2
Load-time and run time Linking, 16-9
Local context, 11-16
Local data
allocation in assembly-language procedure, 14-24
Logical AND operator (&&), 10-25
Logical OR operator (| |), 10-25
LOGICAL*2 type, 14-35
LOGICAL*4 type, 14-35
Long data types
byte ordering on various machines, 15-32
Long double float data type
C run-time library support, 9-5
size, 15-5
support for, 9-1
longjmp, 5-3
LONG_MAX, 15-5
LONG_MIN, 15-5
Loop optimization
disabling unsafe, 6-16
performing, 6-15
Loops, 6-15
Loop_opt pragma, 6-15

M

Macros
creating trouble-free in _asm blocks, 8-18
in-line assembly, 8-5
NULL, C-20
portability issues, 15-28
portable character-classification, 15-20
predefined, A-20
predefined migration issues, A-20
problems with toupper, 15-29
PWB macros, 12-6
MAKE
differences from NMAKE, 10-33
Makefiles
creating in PWB, 3-43
defined, 10-3
examples of, G-1
how to create, 10-1
how to use NMAKE type in PWB, 5-35
notes, 5-34
PWB, 5-34
two types, 5-34
MALLOCMODE.Asm
changing defaults, 5-31
Manifest constants
NULL, 7-9
used in Help files, 11-29

Mantissa
  length of for floating-point data types, 9-2
  storage, 9-3
MASM
  advantages over in-line assembler, 14-21
  invoking automatically from CL, 5-3
  using, 14-21
Math coprocessor package, 9-6
Math intrinsics, 6-10
Math library functions
  summary of locations, A-30
Math packages
  alternate math, 9-6
  coprocessor, 9-6
  emulator, 9-5
  floating-point, 9-5
MAXVAL
  heap size subparameter, 5-28
MC0.Obj
  using for primary heap and stack sharing overflow, 5-30
  using for stack sharing, 5-29
MC0HEAP.Obj
  using for primary heap only, 5-30
MCBIND
  command form, 5-14
  compiler option to use, 5-6
  overview, 1-2
  sample command form, 5-20
  using, 5-6
  using to build for CodeView, 13-4
MCMEMORY.Obj
  adding LDTs, 5-31
  using, 5-32
Memory
  _huge, 5-32
  allocating zero, C-27
  changing addresses in CodeView, 13-28
  displaying in CodeView, 13-25
  handling insufficient memory situations, 15-19
  managing, 7-1
  RAM requirements, 2-1
Memory models
  advantages and disadvantages of using, 7-5
  customizing, 7-20
  default, 5-3
  how choosing affects default pointer size, 14-12
  huge, 7-8
  library support for customized, 7-25
  limitations on code and data size, 7-8
  limits, 7-6
  migration issues, A-11
  mixing, 7-11
  mixing rules, 7-13
  selecting, 7-6
  standard, 7-5
  standard with custom equivalents, 7-21
Menus
  PWB, 3-8
  using QuickHelp, 4-7
Microsoft C Keywords with DLLs, 16-18
Migrating from BTOC C to CTOS Microsoft C
  program changes, A-2

Migration
BTOS C to CTOS, A-1
case translation issues, A-7
coding convention issues, A-14
COMDEF OMF vs. PUBDEF records, A-20
compiler option issues, A-15
documented BTOS C function with substitutes, A-4
documented BTOS C functions without substitutes, A-3
DS allocation, A-21
extended keywords, A-13
function character limits, A-15
function issues, A-2
function prototypes, A-14
function status from BTOS C to CTOS, A-22
global variable issues, A-20
header file conversions, A-7
header file differences: BTOS to CTOS, A-6
huge memory model, A-11
initialization issues, A-15
keyword issues, A-12
library functions, A-22
math library function, A-30
memory model issues, A-11
mixed-language programming, A-14
predefined macro issues, A-20
replacing ctos.h references with ctos.lib.h, A-6
undocumented BTOS C functions, A-2
undocumented BTOS C functions with work-arounds, A-5
MIN.OBJ
defined, 5-22
Minimally formatted ASCII text file, 11-30
Mixed-language programming
parameter passing, 14-16
Mixed language calls
executing, 14-12
how they work, 14-1
making, 14-1
syntax, 14-2
Mixed-language naming convention requirements, 14-4
Mixed-language programming, 14-1
MLIB
command form summary, B-28
command form syntax, 5-26
command line options, B-28
MLINK
/NOD described, 5-17
/PACKC described, 6-26
automatic operation, 5-6
command form, 5-7
command form summary, B-29
command line syntax, 5-7
compiler option used to disable automatic linking, 5-3
config file parameters, 6-25
controlling automatic linking, 5-9
default libraries, 5-17
default libraries when used manually, 5-13
defaults related to MCBIND fields, 5-12

MLINK (continued)
how to use, 5-7
manual operation, 5-6
options correlating to linker config file options, 5-17
overview, 1-2
using, 5-6
using a response file, 5-12
using to build for CodeView, 13-4
MODEL directive, 8-12
Modules
naming, 7-27
Modulus operator (%), 10-25
Monitors
how Help text appears, 11-26
Multibyte characters
ANSI standard and CTOS implementation, C-5
Multiplication operator (*), 10-25
Multithread
creating CTOS III multithread applications, 16-1
Include files, 16-5
Multithread C Library
LLIBCMT.LIB, 16-3
Multithread library compile option /MT, 16-4
Multithread program, 16-1
Mutithread options, 16-4

N

Naming conventions
/Gc option and affects on Pascal programs, 14-13
data types for C, FORTRAN, and Pascal, 14-34
defaults for languages, 14-33
mixed-language identifiers, 14-4
with mixed languages, 14-9
Naming modules and segments, 7-27
Near heap memory
controlling placement of in DGroup, 5-27
sizing, 5-27
three options for placing in DGroup, 5-28
using primary area only, 5-30
Near pointers
how they work, 7-2
Nested comments, A-19
New Features, 1-11
Newline character
placing in a description file, 10-9
terminating, C-22
Newline escape sequence, 10-7

NMAKE
Description Files, 10-6
command example, 10-3
command form summary, B-38
command line options, 10-28
differences from MAKE, 10-33
directive operators, 10-25
environment variables recognized, B-38
indentation of command lines, 10-6
in-line files, 10-30
line-continuation character (\), 10-6
managing development projects with, 10-1
operations sequence, 10-31
overview, 1-3
spawning vs. submit files, 5-35
two methods of executing makefiles, 5-35
using with Batch, 5-35
NMAKE description files
example, creating a library, G-21
example, creating a library version string, G-4
example, creating a PWB extension, G-6
example, creating a run file, G-12
example, creating Help files, G-2
example, creating run files, libraries, and a system service, G-30
NMAKE directives
rules for using, 10-26
NMAKE inference rules
extension search paths, 10-21
precedence, 10-22
predefined, 10-22
user defined, 10-20
NMAKE macros
case distinction, 10-15
environment variable effects, 10-19
in description files, 10-11
inherited from environment variables, 10-19
invoking, 10-13
precedence among definitions, 10-19
predefined list, 10-14
redefining inherited, 10-19
rules for handling spaces, 10-12
substitution within, 10-17
syntax, 10-12
user-defined, 10-12
NO87 environment variable
using, 9-15
NOP instruction, 8-5
Normalized value, C-11
Null characters, C-23
NULL definition, 15-17
NULL macro, C-20
NULL pointers, 7-9
portability, 15-17
Number sign (#), 10-11
Numeric data representation across languages, 14-33
Numerical ranges, D-4
Numerical values defined in FLOAT.H, D-6

O

Obj Flavor object modules
REALHUGE.Obj, 5-32
One's complement arithmetic, 15-15
Online Help, 4-1
Optimization
/Ol, 6-15
from PWB, 6-2
linker and packing code, 6-28
Open files
increasing the maximum number, 2-15
Operating systems
compatibility, 2-1
Operations sequence
NMAKE, 10-31
Operators
directive for NMAKE, 10-25
using in _asm blocks, 8-9
Optimization
/G0, /G1, and /G2 described, 6-21
/Gc described, 6-30
/Gd described, 6-29
/Gr described, 6-30
/Gs described, 6-18
/Oa and /Ow described, 6-12
/Oc and /Og described, 6-20
/Oe described, 6-18
/Oe incompatible with /r, 6-20
/Oi described, 6-8
/On described, 6-16
/Op described, 6-20
/Or disables /Oc, 6-20
/Or incompatible with /Og, 6-20
/Ot and /Os described, 6-8
Optimization (continued
/Ou and /Oy described, 6-23
/Ox described, 6-22
/Oz described, 6-16
achieving consistent floating-point results, 6-20
achieving the smallest code, 6-23
aggressive, 6-16
argument-passing convention, 6-31
assume no aliasing, 6-12
C calling convention, 6-29
common subexpression elimination, 6-5
constant propagation, 6-6
constant subexpression elimmination, 9-10
controlling from the command line, 6-3
controlling with pragmas, 6-3
customizing, 6-7
dead-store elimination, 6-6
default, 6-5
default compiler switches, 6-7
disabling unsafe loop, 6-16
disabling with switches and pragmas, 6-5
effects of _asm blocks, 8-20
enabling aggressive, 6-16
enabling common subexpression, 6-20
enabling far call, 6-25
for debug compiles, 6-2
for release compiles, 6-2
for speed, 6-3
FORTRAN/Pascal calling convention, 6-30

Optimization (continued)
function calling conventions, 6-29
function prolog and epilog, 6-23
generating intrinsic function, 6-8
global incompatibility with setjmp and longjmp, 5-3
global register allocation, 6-18
linker, 6-24
linker and enabling far call, 6-25
linker options that control, 6-24
list of compiler switches, 6-7
loops, 6-3
maximum efficiency, 6-22
performing loop, 6-15
register calling convention, 6-30
removing stack probes, 6-18
specifying compiler, 6-1
speed or size, 6-8
three ways by the compiler, 6-1
troubleshooting, 6-14
types inhibited by in-line assembly code, 8-21
Optimization
using the 80186, 80188, or 80286 processor, 6-21
without aliasing, 6-3
Optimizing C programs, 6-1
Optimizing program size, 6-8
Optimizing program speed, 6-8
Order and alignment issues
structures, 15-8
unions, 15-10
Output redirection, 5-39
Overlays
generating, 6-24
how to create, 5-4
incompatibilities, 5-5
migration, A-21
producing compatible code, A-21
Overview
Browser, 1-7
CodeView, 1-6
compiler and linker, 1-2
CTOS Microsoft C, 1-1
Environment Service, 1-11
Help utilities, 1-9
MLIB, 1-10
NMAKE, 1-3
PWB, 1-5
Task Management Service, 1-7

P

PackCode, 6-28
Packing code, 6-28
Parameter passing
defaults for mixed languages, 14-11
mixed-language programming, 14-9
Parameters
accessing in assembly, 14-27
mixed-language programming, 14-43
passing by reference or value in mixed languages, 14-16
passing defaults for mixed languages, 14-11
passing requirements, 14-9

Parameters (continued)
passing using far reference, 14-10
passing using near reference, 14-10
passing using variable values, 14-10
three methods for passing, 14-9
pArg, 12-21
Partition
creating a QuickHelp, 4-25
Pascal
calling a Pascal program, 14-1
calling conventions, 14-7
conventions, 14-33
data type naming conventions, 14-34
far memory and far reference, 14-10
LSTRING not compatible with C, 14-37
naming conventions, 14-4
string format, 14-36
Passing addresses of common blocks, 14-42
Passing parameters
mixed-language agreement on reference or value, 14-16
using variable values, 14-10
with far reference, 14-10
with near reference, 14-10
perror function, 5-40
perror function messages, C-26
Pointer arithmetic
in huge models, 7-9
Pointer conversion
%hp, C-25
%lp, C-25
%p, C-25
Pointer subtraction
ANSI standards and CTOS implementation, C-12
Pointer values
printing, C-25
reading, C-25
Pointers
advantages of based, 7-30
advantages of near and far, 7-30
and aliasing optimization, 6-12
ANSI standards and CTOS implementation, C-11
based on a pointer, 7-36
based on a segment constant, 7-33
based on a segment variable, 7-34
based on a self segment, 7-38
based on void, 7-38
based problems, 7-15
casting, C-11
casting far to near, 7-13
casting portability, 15-16
checking for undefined in CodeView, 13-30
compiler issues, 6-12
conversions, 7-15
default sizes, 15-19
determining size, 15-16
far, 7-3
how memory model affects size, 14-12
huge, 7-4
huge and migration issues, A-11
keywords, 7-17

Pointers (continued)
near, 7-2
null, 15-17
NULL data, 7-9
null sizing issues, 7-9
passing as arguments in multiple-segment programs, 7-23
passing as arguments to functions, 7-15
passing in mixed-language programming, 14-41
portability, 15-15
problems to avoid, 7-13
relationship to 64K segments, 7-1
rules for declaring variables with keywords, 7-17
setting all to one size with a standard memory model, 7-5
size of generic in bits, 15-18
sizes of generic in a table, 15-18
sizing code in a customized memory model, 7-22
sizing data, 7-22
subtraction, C-12
subtraction of char pointers in character arrays, 15-18
types pushed on the stack, 6-31
types supported, 7-2
Portability
address space issues, 15-19
arithmetic modes, 15-14
bitwise right-shift operations, 15-25
Portability (continued)
byte ordering in words, 15-10
byte ordering issues, 15-31
case sensitivity, 15-26
char data type, 15-3, 15-24
character classifications in character sets, 15-20
character set case translation issues, 15-21
character set issues, 15-20
compiler issues and assumptions, 15-22
CTOS Microsoft C concerns, 15-31
data file issues, 15-30
data type sizing issues, 15-2
environment issues, 15-30
expression evaluation order, 15-27
float data types, 15-5
function and macro argument issues, 15-28
functions with a variable number of arguments, 15-27
hardware issues, 15-2
identifier length and case, 15-25
int and short int data types, 15-3
issues involving _near, _far, _huge, and _based keywords, 7-11
list of nonportable keywords, 15-31
null pointer issues, 15-17
one's complement arithmetic issues, 15-14

Portability (continued)
pointer casting issues, 15-16
pointer issues, 15-15
pointer size issues, 15-16
pointer subtraction issues, 15-16
promotion from shorter types, 15-22
promotion of data types, 15-22
register issues, 15-26
sign extension issues, 15-22
storage alignment issues, 15-12
storage order and alignment issues, 15-7
structure bit fields, 15-12
structure bit fields order of assignment, 15-13
structure issues, 15-8
two's complement arithmetic issues, 15-14
union issues, 15-10
word byte ordering, 15-10
Portable programs
writing, 15-1
Post-optimization, 6-26
Pragma
ANSI standards and CTOS implementation, C-18
check_stack, 6-18
controlling optimization, 6-3
defined, 6-3
how to use, 6-4
intrinsic, 6-10
list of control optimization, 6-3
Pragma (continued)
loop_opt, 6-15
migration, A-13
optimize with e, 15-26
used to maintain correct compiles with setjmp and longjmp, 5-3
Preprocessing directives
ANSI standards and CTOS implementation, C-16
Preserving register values in assembly, 14-25
printf format specifiers, E-1
Printing
from CodeView, 13-31
from QuickHelp, 4-24
Printing pointer values, C-25
Processor arithmetic mode, 15-14
Processor clock cycles for calling sequence, 6-27
Processor compatibility, 6-22
Program limits at run time, D-3
Program size
optimizing, 6-8
Program speed
optimizing, 6-8
Programmer's WorkBench, 3-1
notes, 5-36
with mixed languages, 14-1
Programs
CTOS Microsoft C portability concerns, 15-31
optimizing C, 6-1
writing portable, 15-1
Promotion
from shorter data types, 15-22
of Floating-point types, 9-4

Prototypes
declaring external routines for program documentation, 14-13
functions and address lengths, 14-13
Pseudofiles
as used in PWB, 12-2
Pseudoinstruction
in-line assembly, 8-7
Pseudotargets
.IGNORE:, 10-27
.SILENT, 10-27
.SUFFIXES, 10-27
ptrdiff_t, C-12
PUBDEF
in received object streams, A-20
versus COMDEF OMF record, A-20
PUBLIC directive, 14-23
PWB
assigning keystrokes, 12-4
assigning multiple keystrokes to one function, 12-4
Browse menu, 3-13
building programs, 3-47
building programs from multiple source files, 3-48
changing Editor settings, 3-28
changing keystrokes, 12-4
choosing menu commands, 3-14
closing menus, 3-14
compiling, linking, and running programs, 3-30
creating a build option, 3-41
creating and selecting makefiles, 3-43
PWB (continued)
creating macros, 3-26
customizing the Editor, 3-27
debugging programs, 3-53
debugging with error messages, 3-54
defining a block, 3-21
dialog box Help, 3-18
dialog boxes, 3-15
disabling keystrokes, 12-5
Edit menu, 3-10
editing TOOLS.INI, 12-3
editor settings, 12-2
File menu, 3-9
four methods to customize, 12-1
getting Help, 3-18
Help menu, 3-18
limitations on keystroke assignments, 12-5
Make menu, 3-11
menus, 3-8
modifying keyboard assignments, 3-29
moving around in a source file, 3-20
Options menu, 3-12
overview, 1-5
overview of menu commands, 3-8
overview of setting compiler and linker options, 3-31
parts of a screen, 3-6
pseudofiles, 12-2
quick reference to setting compiler and linker options, 3-52
Run Application, 3-48

PWB (continued)
Run menu, 3-12
running programs, 3-48
sample program, 3-34
saving compiler and linker options as a build option, 3-41
saving source files, 3-35
Search menu, 3-11
searching for and changing text, 3-24
selecting a build option, 3-46
setting anchors, 3-23
setting bookmarks, 3-22
setting compiler options, 3-35
setting linker options, 3-39
setting switches, 12-2
shaded commands, 3-15
starting, 3-2
starting and running CodeView, 3-64
using, 3-1
using advanced editor features, 3-30
using shortcut keys, 3-15
using the Browser, 3-55
using the Editor, 3-19
using windows, 3-4
View menu, 3-10
PWB extensions
arg, 12-21
argData, 12-21
argType, 12-21
assigning keystrokes to, 12-33
building and using, 12-28
cmdTable, 12-16
compiling, 12-29
debugging, 12-35
fMeta, 12-21
PWB extensions (continued)
how they differ from run files, 12-13
linking, 12-31
loading, 12-32
mandatatory elements, 12-16
pArg, 12-21
RADIX10, 12-19
reading a line from a file, 12-24
receiving parameters, 12-21
sample Center.C, 12-14
summary of functions, 12-24
swiDesc, 12-18
swiTable, 12-18
SWI_BOOLEAN, 12-18
SWI_NUMERIC, 12-19
SWI_SPECIAL, 12-18
using to get a file handle, 12-23
writing and building, 12-13
writing function prototypes, 12-16
writing functions D, 12-20
writing the command table, 12-16
writing the initializing function, 12-20
writing the switch table, 12-18
PWB macros
appending commands to, 12-13
argument syntax, 12-8
arguments, 12-9
conditional operators, 12-10
definitions, 12-7
macro responses, 12-8
macros invoking macros, 12-8

PWB macros (continued)
passing arguments to functions, 12-7
procedure to create, 12-11
procedure to record, 12-12
procedure to save permanently, 12-12
recording, 12-12
syntax, 12-6
temporary, 12-11
working with literal text, 12-6
writing, 12-6
PWBRMAKE
command form summary, B-41
command line options, B-41
using, 3-55

Q

Qualifiers
ANSI standards and CTOS implementation, C-15
Question mark (?), 10-7
QuickHelp
accessing error information, 4-23
command form, 4-2
command form summary, B-43
command line options, B-43
copying and pasting, 4-18
creating a partition, 4-25
cross references, 11-27
dialog boxes, 4-15
displaying keyword information, 4-21
dot commands, 11-23
exiting, 4-4
QuickHelp (continued)
formatting flags, 11-26
navigating through screens, 4-15
running, 4-2
starting, 4-2
structure, 4-5
using, 4-1
using hyperlinks, 4-17
using menus, 4-7
using pull-down menus and dialog boxes, 4-7
QuickHelp format text file, 11-22
Quoted file names, C-17

R

RAM
system requirements, 2-1
Range of char values
ANSI standard and CTOS implementation, C-7
Range of integer values
ANSI standard and CTOS implementation, C-8
Ranges
reading, C-25
Re-entrant code with DLLs, 16-16
Reading pointer values, C-25
Reading ranges, C-25
REALHUGE.Obj
defined, 5-22
Records
mixed-language programming, 14-39
Redirection
and starting CodeView, 13-32
input, 5-39
output, 5-39

Register calling convention, 6-29
Register candidates for argument types, 6-31
Register declarations, 15-26
Register variable storage, 8-20
Register variables
  portability, 15-26
Registers
  ANSI standards and CTOS implementation, C-12
  assembly conventions for simple return values, 14-30
  availability, C-12
  change values in CodeView, 13-28
  displaying in CodeView, 13-27
  enabling global allocation, 6-18
  handling variables in CodeView, 13-32
  preserving in assembly, 14-31
  preserving values in assembly procedure, 14-25
  preserving with /Ou, 6-23
  returning ES:BX, 5-37
Remainders
  ANSI standard and CTOS implementation, C-9
Rename function, C-25
Renaming a file with an existing name, C-25
Return values
  assembly register conventions for simple, 14-30
  from assembly-language procedure, 14-30
Returning a value from assembly, 14-30
Rich text format, 11-31
Rich text format RTF, 11-4
Right shift
  ANSI standard and CTOS implementation, C-9
  bitwise operations, 15-24
Right shift operator (>>), 10-25
Routines
  mixed-language call equivalents, 14-3
RTF, 11-31
Run files
  _huge memory support for real mode, 5-21
  creating a GDT protected with MCBIND, 5-24
  creating a minimum sized with MLINK, 5-25
  creating typical, 5-19
  creating with MLINK, 5-2
  linking minimum sized, 5-24
  linking minimum sized with MCBIND, 5-24
  sample CL command form that chains to MLINK, 5-10
  unpredictible PWB build behavior, 5-34
  version 8, 5-17
Run-time library support of long double types, 9-5
Run-time program limits, D-3

S

scanf format specifiers, E-1
SCHAR_MAX, 15-4
SCHAR_MIN, 15-4
Segment overrides
  in-line assembly, 8-5

Segments
adding more far heap, 5-32
code and data, 7-1
far heap, 5-32
LDT-based, 5-31
naming, 7-27
naming conventions for code and data, 7-27
setting up, 7-23
specifying code and data, 7-29
Sequence point, 15-27
setjmp, 5-3
Setting the data threshold, 7-26
Setting up the assembly procedure, 14-23
Short data types
byte ordering on various machines, 15-32
Short int data type
size, 15-3
SHRT_MAX, 15-4
SHRT_MIN, 15-4
Sign extension
portability issues, 15-22
Signed bitwise operations
ANSI standard and CTOS implementation, C-9
Significant characters without external linkage
ANSI standard and CTOS implementation, C-4
Size
basic data types, 15-2
char data type, 15-3
default pointers, 15-19
determining pointer, 15-16
determining with the sizeof operator, 15-2
double float data type, 15-5
Size (continued)
float data type, 15-5
int data type, 15-3
long double float data type, 15-5
of generic pointers in bits, 15-18
pointers and memory models, 14-12
short int data type, 15-3
sizeof operator
example of using, 15-10
using, 15-2
Size_t type, C-11
Slash (/), 11-6
Square brackets ([ ]), 10-26
Stack
adjusting, 14-8
adjustment convention, 6-32
data types pushed onto, 6-31
how languages push parameters onto, 14-8
mixed-language calls and responsibility for adjusting, 14-7
NDP , 6-32
passing parameters in mixed-language programming, 14-8
setting up the frame in assembly, 14-24
sharing memory with near heap, 5-29
sharing with near heap overflow, 5-30
Stack frame
example, 14-28
how assembly functions use, 14-27
removing generation code with /Oy, 6-24

Stack overflow, 6-18
Stack probes
  removing, 6-18
Stand-alone DLL files, 16-21
Stand-alone libraries, 16-11
Startup object modules
  combining MC0.Obj or MC0Heap with stack and heap specifications, 5-28
Startup object modules
  architecture and startup object modules, 5-27
  defined, 5-21
  Flavor Object Modules, 5-22
  MC0HEAP.Obj and near heap, 5-30
  requirements, 5-20
  using MC0.Obj for stack sharing, 5-29
Statements
  ANSI standards and CTOS implementation, C-15
STD instruction, 8-14
Storage alignment
  of data types, 15-12
Storage order and alignment
  portability, 15-7
Streams
  increasing the number of, 2-16
strerror function, C-28
STRING.h
  contents, 7-14
Strings
  C format, 14-35
  FORTRAN format, 14-36
  handled by C, FORTRAN, and Pascal, 14-35
  Pascal format, 14-36
  passing C to FORTRAN and Pascal, 14-36
  passing Pascal to C, 14-37
Strong typing, 6-31
Structures
  alignment of members, A-19
  ANSI standards and CTOS implementation, C-13
  bit fields in, 15-12
  bit fields order of assignment, 15-13
  displaying in CodeView, 13-21
  mixed-language programming, 14-39
  nameless, 5-40
  non-portable code example, 15-8
  order and alignment issues, 15-8
  passing across
mixed-language interface, 14-40
  portability assumptions, 15-8
  pushed on the stack, 6-31
  reading and writing, 15-12
  size and alignment of bit fields, 15-13
  specifying CTOS system, A-19
  storage methods of C, FORTRAN, and Pascal, 14-39
Subexpression elimination, 6-5
Subexpression optimization
  enabling common, 6-20
Subtraction
  pointer portability issues, 15-16
Subtraction operator (-), 10-25
Summary command button, 4-22
Switch statements
  limits, C-15
Switches
  setting in PWB, 12-2

Syntax rules for keywords, 14-14
system function, C-28

T

Targets
  defined, 10-2
  how NMAKE updates, 10-32
  specifying in multiple description blocks, 10-10
Task Management Service
  installing hard disk, 2-11
  installing memory, 3-2
  overview, 1-7
  starting, 13-7
Text files
  truncation, C-23
Text mode, 5-41
Thread stacks, 16-8
Threads, 16-2
Time
  default, C-19
Time zone, C-30
time_t, A-5
TOOLS.INI
  customizing CodeView, 13-33
  editing in PWB, 12-3
  for CodeView, 13-6
  for PWB, 12-2
Topic text, 11-2
toupper
  problems inplementing as a macro, 15-29
Translation
  ANSI standards and CTOS implementation, C-2
Truncation of floating-point values
  ANSI standard and CTOS implementation, C-11
Two's complement arithmetic, 15-15

U

UCHAR_MAX, 15-4
UINT_MAX, 15-5
Underflow of floating-point values, C-22
Underscore (_), 14-4
Undocumented BTOS C functions
  migration issues, A-2
Unformatted ASCII text file, 11-4
Unions
  ANSI standards and CTOS implementation, C-13
  nameless, 5-40
  non-portable code example, 15-10
  order and alignment issues, 15-10
  portability assumptions, 15-8
  pushed on the stack, 6-31
Unrepresented character constants
  ANSI standard and CTOS implementation, C-6
Unsafe loop optimizations
  disabling, 6-16
Up command button, 4-17

Use alternate math compiler
option, 9-13
User-defined types
mixed-language
programming, 14-39
USHRT_MAX, 15-4

V

Values
range of integer, C-8
Variables
based on a segment constant,
7-33
changing in CodeView, 13-20
declaring as floating-point
data types, 9-2
declaring based, 7-32
displaying in CodeView, 13-19
floating-point precision and
storage, 9-4
promoting floating-point, 9-4
register, 15-26
using based, 7-30
Volume name
size, C-24

W

WhenLoaded, 12-20
Wide characters
ANSI standard and CTOS
implementation, C-6
Wildcard characters, 10-7
Word
byte ordering, 15-10
Writing a multithread program,
16-7
Writing Assembly-language
procedures, 14-22
Writing portable programs
hardware assumptions, 15-2
programming assumptions,
15-1

X

xLIBCP.LIB
defined, 5-23
xLIBFP.LIB
defined, 5-24

Z

Zero-length files, C-24

This Page is Blank

Do Not Print

43931609-000